



PKCS #11 Cryptographic Token Interface Base Specification Version 3.0

Committee Specification Draft 01 /
Public Review Draft 01

29 May 2019

This version:

<https://docs.oasis-open.org/pkcs11/pkcs11-base/v3.0/csprd01/pkcs11-base-v3.0-csprd01.docx>

(Authoritative)

<https://docs.oasis-open.org/pkcs11/pkcs11-base/v3.0/csprd01/pkcs11-base-v3.0-csprd01.html>

<https://docs.oasis-open.org/pkcs11/pkcs11-base/v3.0/csprd01/pkcs11-base-v3.0-csprd01.pdf>

Previous version:

N/A

Latest version:

<https://docs.oasis-open.org/pkcs11/pkcs11-base/v3.0/pkcs11-base-v3.0.docx> (Authoritative)

<https://docs.oasis-open.org/pkcs11/pkcs11-base/v3.0/pkcs11-base-v3.0.html>

<https://docs.oasis-open.org/pkcs11/pkcs11-base/v3.0/pkcs11-base-v3.0.pdf>

Technical Committee:

OASIS PKCS 11 TC

Chairs:

Tony Cox (tony.cox@cryptsoft.com), Cryptsoft Pty Ltd

Robert Relyea (rrelyea@redhat.com), Red Hat

Editors:

Chris Zimman (chris@wmp.com), Individual

Dieter Bong (dieter.bong@utimaco.com), Utimaco IS GmbH

Additional artifacts:

This prose specification is one component of a Work Product that also includes:

- PKCS #11 header files:
<https://docs.oasis-open.org/pkcs11/pkcs11-base/v3.0/csprd01/include/pkcs11-v3.0/>

Related work:

This specification replaces or supersedes:

- *PKCS #11 Cryptographic Token Interface Base Specification Version 2.40*. Edited by Robert Griffin and Tim Hudson. Latest version. <http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/pkcs11-base-v2.40.html>.

This specification is related to:

- *PKCS #11 Cryptographic Token Interface Profiles Version 3.0*. Edited by Tim Hudson. Latest version. <https://docs.oasis-open.org/pkcs11/pkcs11-profiles/v3.0/pkcs11-profiles-v3.0.html>.
- *PKCS #11 Cryptographic Token Interface Current Mechanisms Specification Version 3.0*. Edited by Chris Zimman and Dieter Bong. Latest version. <https://docs.oasis-open.org/pkcs11/pkcs11-curr/v3.0/pkcs11-curr-v3.0.html>.

- PKCS #11 Cryptographic Token Interface Historical Mechanisms Specification Version 3.0. Edited by Chris Zimman and Dieter Bong. Latest version. <https://docs.oasis-open.org/pkcs11/pkcs11-hist/v3.0/pkcs11-hist-v3.0.html>.

Abstract:

This document defines data types, functions and other basic components of the PKCS #11 Cryptoki interface.

Status:

This document was last revised or approved by the OASIS PKCS 11 TC on the above date. The level of approval is also listed above. Check the "Latest version" location noted above for possible later revisions of this document. Any other numbered Versions and other technical work produced by the Technical Committee (TC) are listed at https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=pkcs11#technical.

TC members should send comments on this document to the TC's email list. Others should send comments to the TC's public comment list, after subscribing to it by following the instructions at the "Send A Comment" button on the TC's web page at <https://www.oasis-open.org/committees/pkcs11/>.

This specification is provided under the [RF on RAND Terms](#) Mode of the [OASIS IPR Policy](#), the mode chosen when the Technical Committee was established. For information on whether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the TC's web page (<https://www.oasis-open.org/committees/pkcs11/ipr.php>).

Note that any machine-readable content ([Computer Language Definitions](#)) declared Normative for this Work Product is provided in separate plain text files. In the event of a discrepancy between any such plain text file and display content in the Work Product's prose narrative document(s), the content in the separate plain text file prevails.

Citation format:

When referencing this specification the following citation format should be used:

[PKCS11-Base-v3.0]

PKCS #11 Cryptographic Token Interface Base Specification Version 3.0. Edited by Chris Zimman and Dieter Bong. 29 May 2019. OASIS Committee Specification Draft 01 / Public Review Draft 01. <https://docs.oasis-open.org/pkcs11/pkcs11-base/v3.0/csprd01/pkcs11-base-v3.0-csprd01.html>. Latest version: <https://docs.oasis-open.org/pkcs11/pkcs11-base/v3.0/pkcs11-base-v3.0.html>.

Notices

Copyright © OASIS Open 2019. All Rights Reserved.

All capitalized terms in the following text have the meanings assigned to them in the OASIS Intellectual Property Rights Policy (the "OASIS IPR Policy"). The full [Policy](#) may be found at the OASIS website.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published, and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this section are included on all such copies and derivative works. However, this document itself may not be modified in any way, including by removing the copyright notice or references to OASIS, except as needed for the purpose of developing any document or deliverable produced by an OASIS Technical Committee (in which case the rules applicable to copyrights, as set forth in the OASIS IPR Policy, must be followed) or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by OASIS or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and OASIS DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

OASIS requests that any OASIS Party or any other party that believes it has patent claims that would necessarily be infringed by implementations of this OASIS Committee Specification or OASIS Standard, to notify OASIS TC Administrator and provide an indication of its willingness to grant patent licenses to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification.

OASIS invites any party to contact the OASIS TC Administrator if it is aware of a claim of ownership of any patent claims that would necessarily be infringed by implementations of this specification by a patent holder that is not willing to provide a license to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification. OASIS may include such claims on its website, but disclaims any obligation to do so.

OASIS takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on OASIS' procedures with respect to rights in any document or deliverable produced by an OASIS Technical Committee can be found on the OASIS website. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this OASIS Committee Specification or OASIS Standard, can be obtained from the OASIS TC Administrator. OASIS makes no representation that any information or list of intellectual property rights will at any time be complete, or that any claims in such list are, in fact, Essential Claims.

The name "OASIS" is a trademark of [OASIS](#), the owner and developer of this specification, and should be used only to refer to the organization and its official outputs. OASIS welcomes reference to, and implementation and use of, specifications, while reserving the right to enforce its marks against misleading uses. Please see <https://www.oasis-open.org/policies-guidelines/trademark> for above guidance.

Table of Contents

1	Introduction	9
1.1	IPR Policy	9
1.2	Terminology	9
1.3	Definitions	9
1.4	Symbols and abbreviations.....	10
1.5	Normative References	13
1.6	Non-Normative References	14
2	Platform- and compiler-dependent directives for C or C++	16
2.1	Structure packing	16
2.2	Pointer-related macros	16
3	General data types	18
3.1	General information	18
3.2	Slot and token types	19
3.3	Session types	24
3.4	Object types	26
3.5	Data types for mechanisms	30
3.6	Function types	32
3.7	Locking-related types.....	37
4	Objects	41
4.1	Creating, modifying, and copying objects	42
4.1.1	Creating objects	42
4.1.2	Modifying objects.....	43
4.1.3	Copying objects	43
4.2	Common attributes	44
4.3	Hardware Feature Objects.....	44
4.3.1	Definitions.....	44
4.3.2	Overview.....	44
4.3.3	Clock.....	45
4.3.3.1	Definition	45
4.3.3.2	Description.....	45
4.3.4	Monotonic Counter Objects.....	45
4.3.4.1	Definition	45
4.3.4.2	Description.....	45
4.3.5	User Interface Objects.....	45
4.3.5.1	Definition	45
4.3.5.2	Description.....	46
4.4	Storage Objects	46
4.4.1	The CKA_UNIQUE_ID attribute	47
4.5	Data objects	48
4.5.1	Definitions.....	48
4.5.2	Overview.....	48
4.6	Certificate objects	48
4.6.1	Definitions.....	48
4.6.2	Overview.....	48

4.6.3 X.509 public key certificate objects	49
4.6.4 WTLS public key certificate objects.....	51
4.6.5 X.509 attribute certificate objects	52
4.7 Key objects	53
4.7.1 Definitions	53
4.7.2 Overview.....	53
4.8 Public key objects	54
4.9 Private key objects.....	56
4.9.1 RSA private key objects	58
4.10 Secret key objects	58
4.11 Domain parameter objects.....	60
4.11.1 Definitions	60
4.11.2 Overview.....	61
4.12 Mechanism objects	61
4.12.1 Definitions	61
4.12.2 Overview.....	61
4.13 Profile objects	62
4.13.1 Definitions.....	62
4.13.2 Overview.....	62
5 Functions	63
5.1 Function return values	66
5.1.1 Universal Cryptoki function return values.....	67
5.1.2 Cryptoki function return values for functions that use a session handle	67
5.1.3 Cryptoki function return values for functions that use a token	68
5.1.4 Special return value for application-supplied callbacks	68
5.1.5 Special return values for mutex-handling functions	68
5.1.6 All other Cryptoki function return values	68
5.1.7 More on relative priorities of Cryptoki errors	73
5.1.8 Error code “gotchas”	74
5.2 Conventions for functions returning output in a variable-length buffer	74
5.3 Disclaimer concerning sample code	75
5.4 General-purpose functions	75
5.4.1 C_Initialize	75
5.4.2 C_Finalize.....	76
5.4.3 C_GetInfo	76
5.4.4 C_GetFunctionList.....	77
5.4.5 C_GetInterfaceList	78
5.4.6 C_GetInterface	79
5.5 Slot and token management functions	81
5.5.1 C_GetSlotList	81
5.5.2 C_GetSlotInfo	82
5.5.3 C_GetTokenInfo	83
5.5.4 C_WaitForSlotEvent.....	84
5.5.5 C_GetMechanismList	85
5.5.6 C_GetMechanismInfo.....	86
5.5.7 C_InitToken	86

5.5.8 C_InitPIN	88
5.5.9 C_SetPIN.....	88
5.6 Session management functions.....	89
5.6.1 C_OpenSession	90
5.6.2 C_CloseSession	90
5.6.3 C_CloseAllSessions	91
5.6.4 C_GetSessionInfo	92
5.6.5 C_SessionCancel.....	92
5.6.6 C_GetOperationState	93
5.6.7 C_SetOperationState	95
5.6.8 C_Login	97
5.6.9 C_LoginUser.....	98
5.6.10 C_Logout.....	99
5.7 Object management functions	100
5.7.1 C_CreateObject.....	100
5.7.2 C_CopyObject	102
5.7.3 C_DestroyObject	103
5.7.4 C_GetObjectSize.....	104
5.7.5 C_GetAttributeValue	105
5.7.6 C_SetAttributeValue	107
5.7.7 C_FindObjectsInit.....	107
5.7.8 C_FindObjects.....	108
5.7.9 C_FindObjectsFinal	109
5.8 Encryption functions	109
5.8.1 C_EncryptInit.....	109
5.8.2 C_Encrypt.....	110
5.8.3 C_EncryptUpdate	111
5.8.4 C_EncryptFinal.....	111
5.9 Message-based encryption functions	113
5.9.1 C_MessageEncryptInit	113
5.9.2 C_EncryptMessage	114
5.9.3 C_EncryptMessageBegin.....	114
5.9.4 C_EncryptMessageNext.....	115
5.9.5 C_EncryptMessageFinal	116
5.10 Decryption functions	118
5.10.1 C_DecryptInit.....	118
5.10.2 C_Decrypt.....	118
5.10.3 C_DecryptUpdate	119
5.10.4 C_DecryptFinal.....	119
5.11 Message-Based Decryption Functions.....	121
5.11.1 C_MessageDecryptInit	121
5.11.2 C_DecryptMessage	122
5.11.3 C_DecryptMessageBegin.....	123
5.11.4 C_DecryptMessageNext	123
5.11.5 C_MessageDecryptFinal	124
5.12 Message digesting functions	124

5.12.1 C_DigestInit	124
5.12.2 C_Digest	125
5.12.3 C_DigestUpdate	125
5.12.4 C_DigestKey	126
5.12.5 C_DigestFinal	126
5.13 Signing and MACing functions	127
5.13.1 C_SignInit	127
5.13.2 C_Sign	128
5.13.3 C_SignUpdate	128
5.13.4 C_SignFinal	129
5.13.5 C_SignRecoverInit	130
5.13.6 C_SignRecover	130
5.14 Message-Based Signing and MACing Functions	131
5.14.1 C_MessageSignInit	131
5.14.2 C_SignMessage	132
5.14.3 C_SignMessageBegin	133
5.14.4 C_SignMessageNext	133
5.14.5 C_MessageSignFinal	134
5.15 Functions for Verifying Signatures and MACs	134
5.15.1 C_VerifyInit	134
5.15.2 C_Verify	135
5.15.3 C_VerifyUpdate	135
5.15.4 C_VerifyFinal	136
5.15.5 C_VerifyRecoverInit	137
5.15.6 C_VerifyRecover	137
5.16 Message-Based Functions for Verifying Signatures and MACs	138
5.16.1 C_MessageVerifyInit	138
5.16.2 C_VerifyMessage	139
5.16.3 C_VerifyMessageBegin	140
5.16.4 C_VerifyMessageNext	140
5.16.5 C_MessageVerifyFinal	141
5.17 Dual-function cryptographic functions	141
5.17.1 C_DigestEncryptUpdate	141
5.17.2 C_DecryptDigestUpdate	144
5.17.3 C_SignEncryptUpdate	147
5.17.4 C_DecryptVerifyUpdate	149
5.18 Key management functions	152
5.18.1 C_GenerateKey	152
5.18.2 C_GenerateKeyPair	153
5.18.3 C_WrapKey	155
5.18.4 C_UnwrapKey	156
5.18.5 C_DeriveKey	158
5.19 Random number generation functions	160
5.19.1 C_SeedRandom	160
5.19.2 C_GenerateRandom	160
5.20 Parallel function management functions	161

5.20.1 C_GetFunctionStatus	161
5.20.2 C_CancelFunction	161
5.21 Callback functions.....	161
5.21.1 Surrender callbacks.....	162
5.21.2 Vendor-defined callbacks	162
6 PKCS #11 Implementation Conformance	163
Appendix A. Acknowledgments	164
Appendix B. Manifest constants	167
Appendix C. Revision History	168

1 Introduction

This document describes the basic PKCS#11 token interface and token behavior.

The PKCS#11 standard specifies an application programming interface (API), called “Cryptoki,” for devices that hold cryptographic information and perform cryptographic functions. Cryptoki follows a simple object based approach, addressing the goals of technology independence (any kind of device) and resource sharing (multiple applications accessing multiple devices), presenting to applications a common, logical view of the device called a “cryptographic token”.

This document specifies the data types and functions available to an application requiring cryptographic services using the ANSI C programming language. The supplier of a Cryptoki library implementation typically provides these data types and functions via ANSI C header files. Generic ANSI C header files for Cryptoki are available from the PKCS#11 web page. This document and up-to-date errata for Cryptoki will also be available from the same place.

Additional documents may provide a generic, language-independent Cryptoki interface and/or bindings between Cryptoki and other programming languages.

Cryptoki isolates an application from the details of the cryptographic device. The application does not have to change to interface to a different type of device or to run in a different environment; thus, the application is portable. How Cryptoki provides this isolation is beyond the scope of this document, although some conventions for the support of multiple types of device will be addressed here and possibly in a separate document.

Details of cryptographic mechanisms (algorithms) may be found in the associated PKCS#11 Mechanisms documents.

1.1 IPR Policy

This specification is provided under the [RF on RAND Terms](#) Mode of the [OASIS IPR Policy](#), the mode chosen when the Technical Committee was established. For information on whether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the TC's web page (<https://www.oasis-open.org/committees/pkcs11/ipr.php>).

1.2 Terminology

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in [\[RFC2119\]](#).

1.3 Definitions

For the purposes of this standard, the following definitions apply:

API	Application programming interface.
Application	Any computer program that calls the Cryptoki interface.
ASN.1	Abstract Syntax Notation One, as defined in X.680.
Attribute	A characteristic of an object.
BER	Basic Encoding Rules, as defined in X.690.
CBC	Cipher-Block Chaining mode, as defined in FIPS PUB 81.
Certificate	A signed message binding a subject name and a public key, or a subject name and a set of attributes.
CMS	Cryptographic Message Syntax (see RFC 5652)

Cryptographic Device	A device storing cryptographic information and possibly performing cryptographic functions. May be implemented as a smart card, smart disk, PCMCIA card, or with some other technology, including software-only.
Cryptoki	The Cryptographic Token Interface defined in this standard.
Cryptoki library	A library that implements the functions specified in this standard.
DER	Distinguished Encoding Rules, as defined in X.690.
DES	Data Encryption Standard, as defined in FIPS PUB 46-3.
DSA	Digital Signature Algorithm, as defined in FIPS PUB 186-4.
EC	Elliptic Curve
ECB	Electronic Codebook mode, as defined in FIPS PUB 81.
IV	Initialization Vector.
MAC	Message Authentication Code.
Mechanism	A process for implementing a cryptographic operation.
Object	An item that is stored on a token. May be data, a certificate, or a key.
PIN	Personal Identification Number.
PKCS	Public-Key Cryptography Standards.
PRF	Pseudo random function.
PTD	Personal Trusted Device, as defined in MeT-PTD
RSA	The RSA public-key cryptosystem.
Reader	The means by which information is exchanged with a device.
Session	A logical connection between an application and a token.
Slot	A logical reader that potentially contains a token.
SSL	The Secure Sockets Layer 3.0 protocol.
Subject Name	The X.500 distinguished name of the entity to which a key is assigned.
SO	A Security Officer user.
TLS	Transport Layer Security.
Token	The logical view of a cryptographic device defined by Cryptoki.
User	The person using an application that interfaces to Cryptoki.
UTF-8	Universal Character Set (UCS) transformation format (UTF) that represents ISO 10646 and UNICODE strings with a variable number of octets.
WIM	Wireless Identification Module.
WTLS	Wireless Transport Layer Security.

1.4 Symbols and abbreviations

The following symbols are used in this standard:

Table 1, Symbols

Symbol	Definition
N/A	Not applicable
R/O	Read-only
R/W	Read/write

82 The following prefixes are used in this standard:

83 *Table 2, Prefixes*

Prefix	Description
C_	Function
CK_	Data type or general constant
CKA_	Attribute
CKC_	Certificate type
CKD_	Key derivation function
CKF_	Bit flag
CKG_	Mask generation function
CKH_	Hardware feature type
CKK_	Key type
CKM_	Mechanism type
CKN_	Notification
CKO_	Object class
CKP_	Pseudo-random function
CKS_	Session state
CKR_	Return value
CKU_	User type
CKZ_	Salt/Encoding parameter source
h	a handle
ul	a CK_ULONG
p	a pointer
pb	a pointer to a CK_BYTE
ph	a pointer to a handle
pul	a pointer to a CK_ULONG

84

85 Cryptoki is based on ANSI C types, and defines the following data types:

86

```

87  /* an unsigned 8-bit value */
88  typedef unsigned char CK_BYTE;
89
90  /* an unsigned 8-bit character */
91  typedef CK_BYTE CK_CHAR;
92
93  /* an 8-bit UTF-8 character */
94  typedef CK_BYTE CK_UTF8CHAR;
95
96  /* a BYTE-sized Boolean flag */
97  typedef CK_BYTE CK_BBOOL;
98
99  /* an unsigned value, at least 32 bits long */

```

```

typedef unsigned long int CK_ULONG;

/* a signed value, the same size as a CK_ULONG */
typedef long int CK_LONG;

/* at least 32 bits; each bit is a Boolean flag */
typedef CK_ULONG CK_FLAGS;

```

Cryptoki also uses pointers to some of these data types, as well as to the type void, which are implementation-dependent. These pointer types are:

```

CK_BYTE_PTR      /* Pointer to a CK_BYTE */
CK_CHAR_PTR      /* Pointer to a CK_CHAR */
CK_UTF8CHAR_PTR  /* Pointer to a CK_UTF8CHAR */
CK_ULONG_PTR     /* Pointer to a CK_ULONG */
CK_VOID_PTR      /* Pointer to a void */

```

Cryptoki also defines a pointer to a CK_VOID_PTR, which is implementation-dependent:

```

CK_VOID_PTR_PTR  /* Pointer to a CK_VOID_PTR */

```

In addition, Cryptoki defines a C-style NULL pointer, which is distinct from any valid pointer:

```

NULL_PTR         /* A NULL pointer */

```

It follows that many of the data and pointer types will vary somewhat from one environment to another (e.g., a CK_ULONG will sometimes be 32 bits, and sometimes perhaps 64 bits). However, these details should not affect an application, assuming it is compiled with Cryptoki header files consistent with the Cryptoki library to which the application is linked.

All numbers and values expressed in this document are decimal, unless they are preceded by "0x", in which case they are hexadecimal values.

The **CK_CHAR** data type holds characters from the following table, taken from ANSI C:

Table 3, Character Set

Category	Characters
Letters	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z
Numbers	0 1 2 3 4 5 6 7 8 9
Graphic characters	! " # % & ' () * + , - . / : ; < = > ? [\] ^ _ { } ~
Blank character	' '

The **CK_UTF8CHAR** data type holds UTF-8 encoded Unicode characters as specified in RFC2279. UTF-8 allows internationalization while maintaining backward compatibility with the Local String definition of PKCS #11 version 2.01.

In Cryptoki, the **CK_BBOOL** data type is a Boolean type that can be true or false. A zero value means false, and a nonzero value means true. Similarly, an individual bit flag, **CKF_...**, can also be set (true) or unset (false). For convenience, Cryptoki defines the following macros for use with values of type **CK_BBOOL**:

```

#define CK_FALSE 0
#define CK_TRUE 1

```

For backwards compatibility, header files for this version of Cryptoki also define TRUE and FALSE as (CK_DISABLE_TRUE_FALSE may be set by the application vendor):

```

#ifndef CK_DISABLE_TRUE_FALSE

```

```

143 #ifndef FALSE
144 #define FALSE CK_FALSE
145 #endif
146
147 #ifndef TRUE
148 #define TRUE CK_TRUE
149 #endif
150 #endif
151

```

1.5 Normative References

- [FIPS PUB 46-3]** NIST. *FIPS 46-3: Data Encryption Standard*. October 1999.
URL: <http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf>
- [FIPS PUB 81]** NIST. *FIPS 81: DES Modes of Operation*. December 1980.
URL: <http://csrc.nist.gov/publications/fips/fips81/fips81.htm>
- [FIPS PUB 186-4]** NIST. *FIPS 186-4: Digital Signature Standard*. July, 2013.
URL: <http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf>
- [PKCS11-Curr]** *PKCS #11 Cryptographic Token Interface Current Mechanisms Specification Version 2.40*. Edited by Susan Gleeson and Chris Zimman. 14 April 2015. OASIS Standard. <http://docs.oasis-open.org/pkcs11/pkcs11-curr/v2.40/os/pkcs11-curr-v2.40-os.html>. Latest version: <http://docs.oasis-open.org/pkcs11/pkcs11-curr/v2.40/pkcs11-curr-v2.40.html>.
- [PKCS11-Hist]** *PKCS #11 Cryptographic Token Interface Historical Mechanisms Specification Version 2.40*. Edited by Susan Gleeson and Chris Zimman. 14 April 2015. OASIS Standard. <http://docs.oasis-open.org/pkcs11/pkcs11-hist/v2.40/os/pkcs11-hist-v2.40-os.html>. Latest version: <http://docs.oasis-open.org/pkcs11/pkcs11-hist/v2.40/pkcs11-hist-v2.40.html>.
- [PKCS11-Prof]** *PKCS #11 Cryptographic Token Interface Profiles Version 2.40*. Edited by Tim Hudson. 14 April 2015. OASIS Standard. <http://docs.oasis-open.org/pkcs11/pkcs11-profiles/v2.40/os/pkcs11-profiles-v2.40-os.html>. Latest version: <http://docs.oasis-open.org/pkcs11/pkcs11-profiles/v2.40/pkcs11-profiles-v2.40.html>.
- [PKCS #1]** RSA Laboratories. *RSA Cryptography Standard*. v2.1, June 14, 2002.
URL: <ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf>
- [PKCS #3]** RSA Laboratories. *Diffie-Hellman Key-Agreement Standard*. v1.4, November 1993.
URL: <ftp://ftp.rsasecurity.com/pub/pkcs/doc/pkcs-3.doc>
- [PKCS #5]** RSA Laboratories. *Password-Based Encryption Standard*. v2.0, March 25, 1999
URL: <ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-5v2/pkcs5v2-0.pdf>
- [PKCS #7]** RSA Laboratories. *Cryptographic Message Syntax Standard*. v1.5, November 1993
URL : <ftp://ftp.rsasecurity.com/pub/pkcs/doc/pkcs-7.doc>
- [PKCS #8]** RSA Laboratories. *Private-Key Information Syntax Standard*. v1.2, November 1993.
URL: <ftp://ftp.rsasecurity.com/pub/pkcs/doc/pkcs-8.doc>
- [PKCS11-UG]** *PKCS #11 Cryptographic Token Interface Usage Guide Version 2.40*. Edited by John Leiseboer and Robert Griffin. 16 November 2014. OASIS Committee Note 02. <http://docs.oasis-open.org/pkcs11/pkcs11-ug/v2.40/cn02/pkcs11-ug-v2.40-cn02.html>. Latest version: <http://docs.oasis-open.org/pkcs11/pkcs11-ug/v2.40/pkcs11-ug-v2.40.html>.
- [PKCS #12]** RSA Laboratories. *Personal Information Exchange Syntax Standard*. v1.0, June 1999.

194	[RFC2119]	Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
195		URL: http://www.ietf.org/rfc/rfc2119.txt .
196		
197	[RFC 2279]	F. Yergeau. <i>RFC 2279: UTF-8, a transformation format of ISO 10646</i> Alis Technologies, January 1998.
198		URL: http://www.ietf.org/rfc/rfc2279.txt
199		
200	[RFC 2534]	Masinter, L., Wing, D., Mutz, A., and K. Holtman. <i>RFC 2534: Media Features for Display, Print, and Fax</i> . March 1999.
201		URL: http://www.ietf.org/rfc/rfc2534.txt
202		
203	[RFC 5652]	R. Housley. <i>RFC 5652: Cryptographic Message Syntax</i> . Septmber 2009. URL: http://www.ietf.org/rfc/rfc5652.txt
204		
205	[RFC 5707]	Rescorla, E., "The Keying Material Exporters for Transport Layer Security (TLS)", RFC 5705, March 2010.
206		URL: http://www.ietf.org/rfc/rfc5705.txt
207		
208	[TLS]	[RFC2246] Dierks, T. and C. Allen, "The TLS Protocol Version 1.0", RFC 2246, January 1999. URL: http://www.ietf.org/rfc/rfc2246.txt , superseded by [RFC4346]
209		Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1", RFC 4346, April 2006. URL: http://www.ietf.org/rfc/rfc4346.txt , which was
210		superseded by [TLS12].
211		
212		
213	[TLS12]	[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.
214		URL: http://www.ietf.org/rfc/rfc5246.txt
215		
216	[X.500]	ITU-T. Information Technology — Open Systems Interconnection — The Directory: Overview of Concepts, Models and Services. February 2001. Identical to ISO/IEC 9594-1
217		
218		
219	[X.509]	ITU-T. Information Technology — Open Systems Interconnection — The Directory: Public-key and Attribute Certificate Frameworks. March 2000. Identical to ISO/IEC 9594-8
220		
221		
222	[X.680]	ITU-T. Information Technology — Abstract Syntax Notation One (ASN.1): Specification of Basic Notation. July 2002. Identical to ISO/IEC 8824-1
223		
224	[X.690]	ITU-T. Information Technology — ASN.1 Encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER). July 2002. Identical to ISO/IEC 8825-1
225		
226		
227		

228 1.6 Non-Normative References

229	[ANSI C]	ANSI/ISO. American National Standard for Programming Languages – C. 1990.
230	[CC/PP]	W3C. Composite Capability/Preference Profiles (CC/PP): Structure and Vocabularies. World Wide Web Consortium, January 2004.
231		URL: http://www.w3.org/TR/CCPP-struct-vocab/
232		
233	[CDPD]	Ameritech Mobile Communications et al. Cellular Digital Packet Data System Specifications: Part 406: Airlink Security. 1993.
234		
235	[GCS-API]	X/Open Company Ltd. Generic Cryptographic Service API (GCS-API), Base - Draft 2. February 14, 1995.
236		
237	[ISO/IEC 7816-1]	ISO. Information Technology — Identification Cards — Integrated Circuit(s) with Contacts — Part 1: Physical Characteristics. 1998.
238		
239	[ISO/IEC 7816-4]	ISO. Information Technology — Identification Cards — Integrated Circuit(s) with Contacts — Part 4: Interindustry Commands for Interchange. 1995.
240		
241	[ISO/IEC 8824-1]	ISO. Information Technology-- Abstract Syntax Notation One (ASN.1): Specification of Basic Notation. 2002.
242		
243	[ISO/IEC 8825-1]	ISO. Information Technology—ASN.1 Encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER). 2002.
244		
245		

246	[ISO/IEC 9594-1]	ISO. Information Technology — Open Systems Interconnection — The Directory: Overview of Concepts, Models and Services. 2001.
247		
248	[ISO/IEC 9594-8]	ISO. Information Technology — Open Systems Interconnection — The Directory: Public-key and Attribute Certificate Frameworks. 2001
249		
250	[ISO/IEC 9796-2]	ISO. Information Technology — Security Techniques — Digital Signature Scheme Giving Message Recovery — Part 2: Integer factorization based mechanisms. 2002.
251		
252		
253	[Java MIDP]	Java Community Process. Mobile Information Device Profile for Java 2 Micro Edition. November 2002.
254		URL: http://jcp.org/jsr/detail/118.jsp
255		
256	[MeT-PTD]	MeT. MeT PTD Definition – Personal Trusted Device Definition, Version 1.0, February 2003.
257		URL: http://www.mobiletransaction.org
258		
259	[PCMCIA]	Personal Computer Memory Card International Association. <i>PC Card Standard</i> , Release 2.1., July 1993.
260		
261	[SEC 1]	Standards for Efficient Cryptography Group (SECG). <i>Standards for Efficient Cryptography (SEC) 1: Elliptic Curve Cryptography</i> . Version 1.0, September 20, 2000.
262		
263		
264	[SEC 2]	Standards for Efficient Cryptography Group (SECG). <i>Standards for Efficient Cryptography (SEC) 2: Recommended Elliptic Curve Domain Parameters</i> . Version 1.0, September 20, 2000.
265		
266		
267	[WIM]	WAP. Wireless Identity Module. — WAP-260-WIM-20010712-a. July 2001.
268		URL:
269		http://technical.openmobilealliance.org/tech/affiliates/LicenseAgreement.asp?DocName=/wap/wap-260-wim-20010712-a.pdf
270		
271	[WPKI]	Wireless Application Protocol: Public Key Infrastructure Definition. — WAP-217-WPKI-20010424-a. April 2001.
272		URL:
273		http://technical.openmobilealliance.org/tech/affiliates/LicenseAgreement.asp?DocName=/wap/wap-217-wpki-20010424-a.pdf
274		
275		
276	[WTLS]	WAP. Wireless Transport Layer Security Version — WAP-261-WTLS-20010406-a. April 2001.
277		URL:
278		http://technical.openmobilealliance.org/tech/affiliates/LicenseAgreement.asp?DocName=/wap/wap-261-wtls-20010406-a.pdf
279		
280		
281		

2 Platform- and compiler-dependent directives for C or C++

There is a large array of Cryptoki-related data types that are defined in the Cryptoki header files. Certain packing and pointer-related aspects of these types are platform and compiler-dependent; these aspects are therefore resolved on a platform-by-platform (or compiler-by-compiler) basis outside of the Cryptoki header files by means of preprocessor directives.

This means that when writing C or C++ code, certain preprocessor directives **MUST** be issued before including a Cryptoki header file. These directives are described in the remainder of this section.

Platform specific implementation hints can be found in the pkcs11.h header file.

2.1 Structure packing

Cryptoki structures are packed to occupy as little space as is possible. Cryptoki structures **SHALL** be packed with 1-byte alignment.

2.2 Pointer-related macros

Because different platforms and compilers have different ways of dealing with different types of pointers, the following 6 macros **SHALL** be set outside the scope of Cryptoki:

◆ CK_PTR

CK_PTR is the “indirection string” a given platform and compiler uses to make a pointer to an object. It is used in the following fashion:

```
typedef CK_BYTE CK_PTR CK_BYTE_PTR;
```

◆ CK_DECLARE_FUNCTION

CK_DECLARE_FUNCTION(returnType, name), when followed by a parentheses-enclosed list of arguments and a semicolon, declares a Cryptoki API function in a Cryptoki library. returnType is the return type of the function, and name is its name. It **SHALL** be used in the following fashion:

```
CK_DECLARE_FUNCTION(CK_RV, C_Initialize)(  
    CK_VOID_PTR pReserved  
);
```

◆ CK_DECLARE_FUNCTION_POINTER

CK_DECLARE_FUNCTION_POINTER(returnType, name), when followed by a parentheses-enclosed list of arguments and a semicolon, declares a variable or type which is a pointer to a Cryptoki API function in a Cryptoki library. returnType is the return type of the function, and name is its name. It **SHALL** be used in either of the following fashions to define a function pointer variable, myC_Initialize, which can point to a C_Initialize function in a Cryptoki library (note that neither of the following code snippets actually assigns a value to myC_Initialize):

```
CK_DECLARE_FUNCTION_POINTER(CK_RV, myC_Initialize)(  
    CK_VOID_PTR pReserved  
);
```

or:

```
typedef CK_DECLARE_FUNCTION_POINTER(CK_RV, myC_InitializeType)(
```



```
321     CK_VOID_PTR pReserved
322 );
323 myC_InitializeType myC_Initialize;
```

324 ♦ CK_CALLBACK_FUNCTION

325 CK_CALLBACK_FUNCTION(returnType, name), when followed by a parentheses-enclosed
326 list of arguments and a semicolon, declares a variable or type which is a pointer to an application callback
327 function that can be used by a Cryptoki API function in a Cryptoki library. returnType is the return type of
328 the function, and name is its name. It SHALL be used in either of the following fashions to define a
329 function pointer variable, myCallback, which can point to an application callback which takes arguments
330 args and returns a CK_RV (note that neither of the following code snippets actually assigns a value to
331 myCallback):

```
332 CK_CALLBACK_FUNCTION(CK_RV, myCallback)(args);
333
```

334 or:

```
335 typedef CK_CALLBACK_FUNCTION(CK_RV, myCallbackType)(args);
336 myCallbackType myCallback;
```

337 ♦ NULL_PTR

338 NULL_PTR is the value of a NULL pointer. In any ANSI C environment—and in many others as well—
339 NULL_PTR SHALL be defined simply as 0.

3 General data types

The general Cryptoki data types are described in the following subsections. The data types for holding parameters for various mechanisms, and the pointers to those parameters, are not described here; these types are described with the information on the mechanisms themselves, in Section 12.

A C or C++ source file in a Cryptoki application or library can define all these types (the types described here and the types that are specifically used for particular mechanism parameters) by including the top-level Cryptoki include file, `pkcs11.h`. `pkcs11.h`, in turn, includes the other Cryptoki include files, `pkcs11t.h` and `pkcs11f.h`. A source file can also include just `pkcs11t.h` (instead of `pkcs11.h`); this defines most (but not all) of the types specified here.

When including either of these header files, a source file **MUST** specify the preprocessor directives indicated in Section 2.

3.1 General information

Cryptoki represents general information with the following types:

◆ **CK_VERSION; CK_VERSION_PTR**

CK_VERSION is a structure that describes the version of a Cryptoki interface, a Cryptoki library, or an SSL or TLS implementation, or the hardware or firmware version of a slot or token. It is defined as follows:

```
typedef struct CK_VERSION {  
    CK_BYTE major;  
    CK_BYTE minor;  
} CK_VERSION;
```

The fields of the structure have the following meanings:

major major version number (the integer portion of the version)

minor minor version number (the hundredths portion of the version)

Example: For version 1.0, *major* = 1 and *minor* = 0. For version 2.10, *major* = 2 and *minor* = 10. Table 4 below lists the major and minor version values for the officially published Cryptoki specifications.

Table 4, Major and minor version values for published Cryptoki specifications

Version	major	minor
1.0	0x01	0x00
2.01	0x02	0x01
2.10	0x02	0x0a
2.11	0x02	0x0b
2.20	0x02	0x14
2.30	0x02	0x1e
2.40	0x02	0x28
3.0	0x03	0x00

Minor revisions of the Cryptoki standard are always upwardly compatible within the same major version number.

CK_VERSION_PTR is a pointer to a **CK_VERSION**.

◆ **CK_INFO; CK_INFO_PTR**

CK_INFO provides general information about Cryptoki. It is defined as follows:

```

373 typedef struct CK_INFO {
374     CK_VERSION cryptokiVersion;
375     CK_UTF8CHAR manufacturerID[32];
376     CK_FLAGS flags;
377     CK_UTF8CHAR libraryDescription[32];
378     CK_VERSION libraryVersion;
379 } CK_INFO;
380

```

381 The fields of the structure have the following meanings:

382	<i>cryptokiVersion</i>	Cryptoki interface version number, for compatibility with future
383		revisions of this interface
384	<i>manufacturerID</i>	ID of the Cryptoki library manufacturer. MUST be padded with the
385		blank character (' '). Should <i>not</i> be null-terminated.
386	<i>flags</i>	bit flags reserved for future versions. MUST be zero for this version
387	<i>libraryDescription</i>	character-string description of the library. MUST be padded with the
388		blank character (' '). Should <i>not</i> be null-terminated.
389	<i>libraryVersion</i>	Cryptoki library version number

390 For libraries written to this document, the value of *cryptokiVersion* should match the version of this
391 specification; the value of *libraryVersion* is the version number of the library software itself.

392 **CK_INFO_PTR** is a pointer to a **CK_INFO**.

393 ♦ **CK_NOTIFICATION**

394 **CK_NOTIFICATION** holds the types of notifications that Cryptoki provides to an application. It is defined
395 as follows:

```

396 typedef CK_ULONG CK_NOTIFICATION;
397

```

398 For this version of Cryptoki, the following types of notifications are defined:

```

399 CKN_SURRENDER
400

```

401 The notifications have the following meanings:

402	<i>CKN_SURRENDER</i>	Cryptoki is surrendering the execution of a function executing in a
403		session so that the application may perform other operations. After
404		performing any desired operations, the application should indicate
405		to Cryptoki whether to continue or cancel the function (see Section
406		5.21.1).

407 3.2 Slot and token types

408 Cryptoki represents slot and token information with the following types:

409 ♦ **CK_SLOT_ID; CK_SLOT_ID_PTR**

410 **CK_SLOT_ID** is a Cryptoki-assigned value that identifies a slot. It is defined as follows:

```

411 typedef CK_ULONG CK_SLOT_ID;
412

```

A list of **CK_SLOT_IDs** is returned by **C_GetSlotList**. A priori, any value of **CK_SLOT_ID** can be a valid slot identifier—in particular, a system may have a slot identified by the value 0. It need not have such a slot, however.

CK_SLOT_ID_PTR is a pointer to a **CK_SLOT_ID**.

◆ **CK_SLOT_INFO; CK_SLOT_INFO_PTR**

CK_SLOT_INFO provides information about a slot. It is defined as follows:

```
typedef struct CK_SLOT_INFO {
    CK_UTF8CHAR slotDescription[64];
    CK_UTF8CHAR manufacturerID[32];
    CK_FLAGS flags;
    CK_VERSION hardwareVersion;
    CK_VERSION firmwareVersion;
} CK_SLOT_INFO;
```

The fields of the structure have the following meanings:

<i>slotDescription</i>	character-string description of the slot. MUST be padded with the blank character (' '). MUST NOT be null-terminated.
<i>manufacturerID</i>	ID of the slot manufacturer. MUST be padded with the blank character (' '). MUST NOT be null-terminated.
<i>flags</i>	bits flags that provide capabilities of the slot. The flags are defined below
<i>hardwareVersion</i>	version number of the slot's hardware
<i>firmwareVersion</i>	version number of the slot's firmware

The following table defines the *flags* field:

Table 5, Slot Information Flags

Bit Flag	Mask	Meaning
CKF_TOKEN_PRESENT	0x00000001	True if a token is present in the slot (e.g., a device is in the reader)
CKF_REMOVABLE_DEVICE	0x00000002	True if the reader supports removable devices
CKF_HW_SLOT	0x00000004	True if the slot is a hardware slot, as opposed to a software slot implementing a "soft token"

For a given slot, the value of the **CKF_REMOVABLE_DEVICE** flag *never changes*. In addition, if this flag is not set for a given slot, then the **CKF_TOKEN_PRESENT** flag for that slot is *always* set. That is, if a slot does not support a removable device, then that slot always has a token in it.

CK_SLOT_INFO_PTR is a pointer to a **CK_SLOT_INFO**.

◆ **CK_TOKEN_INFO; CK_TOKEN_INFO_PTR**

CK_TOKEN_INFO provides information about a token. It is defined as follows:

```
typedef struct CK_TOKEN_INFO {
    CK_UTF8CHAR label[32];
```

```

446 CK_UTF8CHAR manufacturerID[32];
447 CK_UTF8CHAR model[16];
448 CK_CHAR serialNumber[16];
449 CK_FLAGS flags;
450 CK_ULONG ulMaxSessionCount;
451 CK_ULONG ulSessionCount;
452 CK_ULONG ulMaxRwSessionCount;
453 CK_ULONG ulRwSessionCount;
454 CK_ULONG ulMaxPinLen;
455 CK_ULONG ulMinPinLen;
456 CK_ULONG ulTotalPublicMemory;
457 CK_ULONG ulFreePublicMemory;
458 CK_ULONG ulTotalPrivateMemory;
459 CK_ULONG ulFreePrivateMemory;
460 CK_VERSION hardwareVersion;
461 CK_VERSION firmwareVersion;
462 CK_CHAR utcTime[16];
463 } CK_TOKEN_INFO;
464

```

465 The fields of the structure have the following meanings:

466	<i>label</i>	application-defined label, assigned during token initialization. MUST be padded with the blank character (' '). MUST NOT be null-terminated.
467		
468		
469	<i>manufacturerID</i>	ID of the device manufacturer. MUST be padded with the blank character (' '). MUST NOT be null-terminated.
470		
471	<i>model</i>	model of the device. MUST be padded with the blank character (' '). MUST NOT be null-terminated.
472		
473	<i>serialNumber</i>	character-string serial number of the device. MUST be padded with the blank character (' '). MUST NOT be null-terminated.
474		
475	<i>flags</i>	bit flags indicating capabilities and status of the device as defined below
476		
477	<i>ulMaxSessionCount</i>	maximum number of sessions that can be opened with the token at one time by a single application (see CK_TOKEN_INFO Note below)
478		
479		
480	<i>ulSessionCount</i>	number of sessions that this application currently has open with the token (see CK_TOKEN_INFO Note below)
481		
482	<i>ulMaxRwSessionCount</i>	maximum number of read/write sessions that can be opened with the token at one time by a single application (see CK_TOKEN_INFO Note below)
483		
484		
485	<i>ulRwSessionCount</i>	number of read/write sessions that this application currently has open with the token (see CK_TOKEN_INFO Note below)
486		
487	<i>ulMaxPinLen</i>	maximum length in bytes of the PIN
488	<i>ulMinPinLen</i>	minimum length in bytes of the PIN
489	<i>ulTotalPublicMemory</i>	the total amount of memory on the token in bytes in which public objects may be stored (see CK_TOKEN_INFO Note below)
490		

Bit Flag	Mask	Meaning
CKF_TOKEN_INITIALIZED	0x00000400	True if the token has been initialized using C_InitToken or an equivalent mechanism outside the scope of this standard. Calling C_InitToken when this flag is set will cause the token to be reinitialized.
CKF_SECONDARY_AUTHENTICATION	0x00000800	True if the token supports secondary authentication for private key objects. (Deprecated; new implementations MUST NOT set this flag)
CKF_USER_PIN_COUNT_LOW	0x00010000	True if an incorrect user login PIN has been entered at least once since the last successful authentication.
CKF_USER_PIN_FINAL_TRY	0x00020000	True if supplying an incorrect user PIN will cause it to become locked.
CKF_USER_PIN_LOCKED	0x00040000	True if the user PIN has been locked. User login to the token is not possible.
CKF_USER_PIN_TO_BE_CHANGED	0x00080000	True if the user PIN value is the default value set by token initialization or manufacturing, or the PIN has been expired by the card.
CKF_SO_PIN_COUNT_LOW	0x00100000	True if an incorrect SO login PIN has been entered at least once since the last successful authentication.
CKF_SO_PIN_FINAL_TRY	0x00200000	True if supplying an incorrect SO PIN will cause it to become locked.
CKF_SO_PIN_LOCKED	0x00400000	True if the SO PIN has been locked. SO login to the token is not possible.
CKF_SO_PIN_TO_BE_CHANGED	0x00800000	True if the SO PIN value is the default value set by token initialization or manufacturing, or the PIN has been expired by the card.
CKF_ERROR_STATE	0x01000000	True if the token failed a FIPS 140-2 self-test and entered an error state.

Exactly what the **CKF_WRITE_PROTECTED** flag means is not specified in Cryptoki. An application may be unable to perform certain actions on a write-protected token; these actions can include any of the following, among others:

- Creating/modifying/deleting any object on the token.
- Creating/modifying/deleting a token object on the token.

- Changing the SO's PIN.

- Changing the normal user's PIN.

The token may change the value of the **CKF_WRITE_PROTECTED** flag depending on the session state to implement its object management policy. For instance, the token may set the **CKF_WRITE_PROTECTED** flag unless the session state is R/W SO or R/W User to implement a policy that does not allow any objects, public or private, to be created, modified, or deleted unless the user has successfully called C_Login.

The **CKF_USER_PIN_COUNT_LOW**, **CKF_USER_PIN_COUNT_LOW**, **CKF_USER_PIN_FINAL_TRY**, and **CKF_SO_PIN_FINAL_TRY** flags may always be set to false if the token does not support the functionality or will not reveal the information because of its security policy.

The **CKF_USER_PIN_TO_BE_CHANGED** and **CKF_SO_PIN_TO_BE_CHANGED** flags may always be set to false if the token does not support the functionality. If a PIN is set to the default value, or has expired, the appropriate **CKF_USER_PIN_TO_BE_CHANGED** or **CKF_SO_PIN_TO_BE_CHANGED** flag is set to true. When either of these flags are true, logging in with the corresponding PIN will succeed, but only the C_SetPIN function can be called. Calling any other function that required the user to be logged in will cause CKR_PIN_EXPIRED to be returned until C_SetPIN is called successfully.

CK_TOKEN_INFO Note: The fields ulMaxSessionCount, ulSessionCount, ulMaxRwSessionCount, ulRwSessionCount, ulTotalPublicMemory, ulFreePublicMemory, ulTotalPrivateMemory, and ulFreePrivateMemory can have the special value CK_UNAVAILABLE_INFORMATION, which means that the token and/or library is unable or unwilling to provide that information. In addition, the fields ulMaxSessionCount and ulMaxRwSessionCount can have the special value CK_EFFECTIVELY_INFINITE, which means that there is no practical limit on the number of sessions (resp. R/W sessions) an application can have open with the token.

It is important to check these fields for these special values. This is particularly true for CK_EFFECTIVELY_INFINITE, since an application seeing this value in the ulMaxSessionCount or ulMaxRwSessionCount field would otherwise conclude that it can't open any sessions with the token, which is far from being the case.

The upshot of all this is that the correct way to interpret (for example) the ulMaxSessionCount field is something along the lines of the following:

```
CK_TOKEN_INFO info;
.
.
if ((CK_LONG) info.ulMaxSessionCount
    == CK_UNAVAILABLE_INFORMATION) {
    /* Token refuses to give value of ulMaxSessionCount */
    .
    .
} else if (info.ulMaxSessionCount == CK_EFFECTIVELY_INFINITE) {
    /* Application can open as many sessions as it wants */
    .
    .
} else {
    /* ulMaxSessionCount really does contain what it should */
    .
    .
}
```

CK_TOKEN_INFO_PTR is a pointer to a CK_TOKEN_INFO.

3.3 Session types

Cryptoki represents session information with the following types:

562 ♦ CK_SESSION_HANDLE; CK_SESSION_HANDLE_PTR

563 **CK_SESSION_HANDLE** is a Cryptoki-assigned value that identifies a session. It is defined as follows:

```
564 typedef CK_ULONG CK_SESSION_HANDLE;  
565
```

566 *Valid session handles in Cryptoki always have nonzero values.* For developers' convenience, Cryptoki
567 defines the following symbolic value:

```
568 CK_INVALID_HANDLE  
569
```

570 **CK_SESSION_HANDLE_PTR** is a pointer to a **CK_SESSION_HANDLE**.

571 ♦ CK_USER_TYPE

572 **CK_USER_TYPE** holds the types of Cryptoki users described in [\[PKCS11-UG\]](#) and, in addition, a
573 context-specific type described in Section 4.9. It is defined as follows:

```
574 typedef CK_ULONG CK_USER_TYPE;  
575
```

576 For this version of Cryptoki, the following types of users are defined:

```
577 CKU_SO  
578 CKU_USER  
579 CKU_CONTEXT_SPECIFIC
```

580 ♦ CK_STATE

581 **CK_STATE** holds the session state, as described in [\[PKCS11-UG\]](#). It is defined as follows:

```
582 typedef CK_ULONG CK_STATE;  
583
```

584 For this version of Cryptoki, the following session states are defined:

```
585 CKS_RO_PUBLIC_SESSION  
586 CKS_RO_USER_FUNCTIONS  
587 CKS_RW_PUBLIC_SESSION  
588 CKS_RW_USER_FUNCTIONS  
589 CKS_RW_SO_FUNCTIONS
```

590 ♦ CK_SESSION_INFO; CK_SESSION_INFO_PTR

591 **CK_SESSION_INFO** provides information about a session. It is defined as follows:

```
592 typedef struct CK_SESSION_INFO {  
593     CK_SLOT_ID slotID;  
594     CK_STATE state;  
595     CK_FLAGS flags;  
596     CK_ULONG ulDeviceError;  
597 } CK_SESSION_INFO;  
598
```

599

600 The fields of the structure have the following meanings:

601 *slotID* ID of the slot that interfaces with the token

602 *state* the state of the session

603 *flags* bit flags that define the type of session; the flags are defined below

604 *ulDeviceError* an error code defined by the cryptographic device. Used for errors

605 not covered by Cryptoki.

606 The following table defines the *flags* field:

607 *Table 7, Session Information Flags*

Bit Flag	Mask	Meaning
CKF_RW_SESSION	0x00000002	True if the session is read/write; false if the session is read-only
CKF_SERIAL_SESSION	0x00000004	This flag is provided for backward compatibility, and should always be set to true

608 CK_SESSION_INFO_PTR is a pointer to a CK_SESSION_INFO.

609 3.4 Object types

610 Cryptoki represents object information with the following types:

611 ♦ CK_OBJECT_HANDLE; CK_OBJECT_HANDLE_PTR

612 CK_OBJECT_HANDLE is a token-specific identifier for an object. It is defined as follows:

```
613 typedef CK_ULONG CK_OBJECT_HANDLE;
614
```

615 When an object is created or found on a token by an application, Cryptoki assigns it an object handle for
616 that application's sessions to use to access it. A particular object on a token does not necessarily have a
617 handle which is fixed for the lifetime of the object; however, if a particular session can use a particular
618 handle to access a particular object, then that session will continue to be able to use that handle to
619 access that object as long as the session continues to exist, the object continues to exist, and the object
620 continues to be accessible to the session.

621 *Valid object handles in Cryptoki always have nonzero values.* For developers' convenience, Cryptoki
622 defines the following symbolic value:

```
623 CK_INVALID_HANDLE
624
```

625 CK_OBJECT_HANDLE_PTR is a pointer to a CK_OBJECT_HANDLE.

626 ♦ CK_OBJECT_CLASS; CK_OBJECT_CLASS_PTR

627 CK_OBJECT_CLASS is a value that identifies the classes (or types) of objects that Cryptoki recognizes.
628 It is defined as follows:

```
629 typedef CK_ULONG CK_OBJECT_CLASS;
630
```

631 Object classes are defined with the objects that use them. The type is specified on an object through the
632 CKA_CLASS attribute of the object.

633 Vendor defined values for this type may also be specified.

```
634 CKO_VENDOR_DEFINED
635
```

636 Object classes **CKO_VENDOR_DEFINED** and above are permanently reserved for token vendors. For
637 interoperability, vendors should register their object classes through the PKCS process.

638 **CK_OBJECT_CLASS_PTR** is a pointer to a **CK_OBJECT_CLASS**.

639 ♦ **CK_HW_FEATURE_TYPE**

640 **CK_HW_FEATURE_TYPE** is a value that identifies a hardware feature type of a device. It is defined as
641 follows:

```
642 typedef CK_ULONG CK_HW_FEATURE_TYPE;  
643
```

644 Hardware feature types are defined with the objects that use them. The type is specified on an object
645 through the **CKA_HW_FEATURE_TYPE** attribute of the object.

646 Vendor defined values for this type may also be specified.

```
647 CKH_VENDOR_DEFINED  
648
```

649 Feature types **CKH_VENDOR_DEFINED** and above are permanently reserved for token vendors. For
650 interoperability, vendors should register their feature types through the PKCS process.

651 ♦ **CK_KEY_TYPE**

652 **CK_KEY_TYPE** is a value that identifies a key type. It is defined as follows:

```
653 typedef CK_ULONG CK_KEY_TYPE;  
654
```

655 Key types are defined with the objects and mechanisms that use them. The key type is specified on an
656 object through the **CKA_KEY_TYPE** attribute of the object.

657 Vendor defined values for this type may also be specified.

```
658 CKK_VENDOR_DEFINED  
659
```

660 Key types **CKK_VENDOR_DEFINED** and above are permanently reserved for token vendors. For
661 interoperability, vendors should register their key types through the PKCS process.

662 ♦ **CK_CERTIFICATE_TYPE**

663 **CK_CERTIFICATE_TYPE** is a value that identifies a certificate type. It is defined as follows:

```
664 typedef CK_ULONG CK_CERTIFICATE_TYPE;  
665
```

666 Certificate types are defined with the objects and mechanisms that use them. The certificate type is
667 specified on an object through the **CKA_CERTIFICATE_TYPE** attribute of the object.

668 Vendor defined values for this type may also be specified.

```
669 CKC_VENDOR_DEFINED  
670
```

671 Certificate types **CKC_VENDOR_DEFINED** and above are permanently reserved for token vendors. For
672 interoperability, vendors should register their certificate types through the PKCS process.

673 ♦ **CK_CERTIFICATE_CATEGORY**

674 **CK_CERTIFICATE_CATEGORY** is a value that identifies a certificate category. It is defined as follows:

```
675 typedef CK_ULONG CK_CERTIFICATE_CATEGORY;  
676
```

677 For this version of Cryptoki, the following certificate categories are defined:

Constant	Value	Meaning
CK_CERTIFICATE_CATEGORY_UNSPECIFIED	0x00000000UL	No category specified
CK_CERTIFICATE_CATEGORY_TOKEN_USER	0x00000001UL	Certificate belongs to owner of the token
CK_CERTIFICATE_CATEGORY_AUTHORITY	0x00000002UL	Certificate belongs to a certificate authority
CK_CERTIFICATE_CATEGORY_OTHER_ENTITY	0x00000003UL	Certificate belongs to an end entity (i.e.: not a CA)

678 ♦ CK_ATTRIBUTE_TYPE

679 **CK_ATTRIBUTE_TYPE** is a value that identifies an attribute type. It is defined as follows:

```
680 typedef CK_ULONG CK_ATTRIBUTE_TYPE;  
681
```

682 Attributes are defined with the objects and mechanisms that use them. Attributes are specified on an
683 object as a list of type, length value items. These are often specified as an attribute template.

684 Vendor defined values for this type may also be specified.

```
685 CKA_VENDOR_DEFINED  
686
```

687 Attribute types **CKA_VENDOR_DEFINED** and above are permanently reserved for token vendors. For
688 interoperability, vendors should register their attribute types through the PKCS process.

689 ♦ CK_ATTRIBUTE; CK_ATTRIBUTE_PTR

690 **CK_ATTRIBUTE** is a structure that includes the type, value, and length of an attribute. It is defined as
691 follows:

```
692 typedef struct CK_ATTRIBUTE {  
693     CK_ATTRIBUTE_TYPE type;  
694     CK_VOID_PTR pValue;  
695     CK_ULONG ulValueLen;  
696 } CK_ATTRIBUTE;  
697
```

698 The fields of the structure have the following meanings:

699 *type* the attribute type

700 *pValue* pointer to the value of the attribute

701 *ulValueLen* length in bytes of the value

702 If an attribute has no value, then *ulValueLen* = 0, and the value of *pValue* is irrelevant. An array of
703 **CK_ATTRIBUTES** is called a “template” and is used for creating, manipulating and searching for objects.
704 The order of the attributes in a template *never* matters, even if the template contains vendor-specific
705 attributes. Note that *pValue* is a “void” pointer, facilitating the passing of arbitrary values. Both the
706 application and Cryptoki library **MUST** ensure that the pointer can be safely cast to the expected type
707 (i.e., without word-alignment errors).

708
709 The constant **CK_UNAVAILABLE_INFORMATION** is used in the *ulValueLen* field to denote an invalid or
710 unavailable value. See **C_GetAttributeValue** for further details.

711
712 **CK_ATTRIBUTE_PTR** is a pointer to a **CK_ATTRIBUTE**.

713 ♦ **CK_DATE**

714 **CK_DATE** is a structure that defines a date. It is defined as follows:

```
715     typedef struct CK_DATE {  
716         CK_CHAR year[4];  
717         CK_CHAR month[2];  
718         CK_CHAR day[2];  
719     } CK_DATE;  
720
```

721 The fields of the structure have the following meanings:

722 *year* the year ("1900" - "9999")

723 *month* the month ("01" - "12")

724 *day* the day ("01" - "31")

725 The fields hold numeric characters from the character set in Table 3, not the literal byte values.

726 When a Cryptoki object carries an attribute of this type, and the default value of the attribute is specified
727 to be "empty," then Cryptoki libraries SHALL set the attribute's *ulValueLen* to 0.

728 Note that implementations of previous versions of Cryptoki may have used other methods to identify an
729 "empty" attribute of type **CK_DATE**, and applications that needs to interoperate with these libraries
730 therefore have to be flexible in what they accept as an empty value.

731 ♦ **CK_PROFILE_ID; CK_PROFILE_ID_PTR**

732 **CK_PROFILE_ID** is an unsigned long value representing a specific token profile. It is defined as follows:

```
733     typedef CK_ULONG CK_PROFILE_ID;  
734
```

735 Profiles are defined in the PKCS #11 Cryptographic Token Interface Profiles document. s. ID's greater
736 than 0xffffffff may cause compatibility issues on platforms that have **CK_ULONG** values of 32 bits, and
737 should be avoided.

738 Vendor defined values for this type may also be specified.

```
739     CKP_VENDOR_ DEFINED  
740
```

741 Profile IDs **CKP_VENDOR_DEFINED** and above are permanently reserved for token vendors. For
742 interoperability, vendors should register their object classes through the PKCS process.

743
744 *Valid Profile IDs in Cryptoki always have nonzero values.* For developers' convenience, Cryptoki defines
745 the following symbolic value:

```
746     CKP_INVALID_ID
```

747 **CK_PROFILE_ID_PTR** is a pointer to a **CK_PROFILE_ID**.

◆ CK_JAVA_MIDP_SECURITY_DOMAIN

CK_JAVA_MIDP_SECURITY_DOMAIN is a value that identifies the Java MIDP security domain of a certificate. It is defined as follows:

```
typedef CK_ULONG CK_JAVA_MIDP_SECURITY_DOMAIN;
```

For this version of Cryptoki, the following security domains are defined. See the Java MIDP specification for further information:

Constant	Value	Meaning
CK_SECURITY_DOMAIN_UNSPECIFIED	0x00000000UL	No domain specified
CK_SECURITY_DOMAIN_MANUFACTURER	0x00000001UL	Manufacturer protection domain
CK_SECURITY_DOMAIN_OPERATOR	0x00000002UL	Operator protection domain
CK_SECURITY_DOMAIN_THIRD_PARTY	0x00000003UL	Third party protection domain

3.5 Data types for mechanisms

Cryptoki supports the following types for describing mechanisms and parameters to them:

◆ CK_MECHANISM_TYPE; CK_MECHANISM_TYPE_PTR

CK_MECHANISM_TYPE is a value that identifies a mechanism type. It is defined as follows:

```
typedef CK_ULONG CK_MECHANISM_TYPE;
```

Mechanism types are defined with the objects and mechanism descriptions that use them.

Vendor defined values for this type may also be specified.

```
CKM_VENDOR_DEFINED
```

Mechanism types **CKM_VENDOR_DEFINED** and above are permanently reserved for token vendors. For interoperability, vendors should register their mechanism types through the PKCS process.

CK_MECHANISM_TYPE_PTR is a pointer to a **CK_MECHANISM_TYPE**.

◆ CK_MECHANISM; CK_MECHANISM_PTR

CK_MECHANISM is a structure that specifies a particular mechanism and any parameters it requires. It is defined as follows:

```
typedef struct CK_MECHANISM {  
    CK_MECHANISM_TYPE mechanism;  
    CK_VOID_PTR pParameter;  
    CK_ULONG ulParameterLen;  
} CK_MECHANISM;
```

The fields of the structure have the following meanings:

mechanism the type of mechanism

779 *pParameter* pointer to the parameter if required by the mechanism

780 *ulParameterLen* length in bytes of the parameter

781 Note that *pParameter* is a “void” pointer, facilitating the passing of arbitrary values. Both the application
782 and the Cryptoki library MUST ensure that the pointer can be safely cast to the expected type (*i.e.*,
783 without word-alignment errors).

784 **CK_MECHANISM_PTR** is a pointer to a **CK_MECHANISM**.

785 ♦ **CK_MECHANISM_INFO; CK_MECHANISM_INFO_PTR**

786 **CK_MECHANISM_INFO** is a structure that provides information about a particular mechanism. It is
787 defined as follows:

```
788 typedef struct CK_MECHANISM_INFO {  
789     CK_ULONG ulMinKeySize;  
790     CK_ULONG ulMaxKeySize;  
791     CK_FLAGS flags;  
792 } CK_MECHANISM_INFO;  
793
```

794 The fields of the structure have the following meanings:

795 *ulMinKeySize* the minimum size of the key for the mechanism (whether this is
796 measured in bits or in bytes is mechanism-dependent)

797 *ulMaxKeySize* the maximum size of the key for the mechanism (whether this is
798 measured in bits or in bytes is mechanism-dependent)

799 *flags* bit flags specifying mechanism capabilities

800 For some mechanisms, the *ulMinKeySize* and *ulMaxKeySize* fields have meaningless values.

801 The following table defines the *flags* field:

802 *Table 8, Mechanism Information Flags*

Bit Flag	Mask	Meaning
CKF_HW	0x00000001	True if the mechanism is performed by the device; false if the mechanism is performed in software
CKF_MESSAGE_ENCRYPT	0x00000002	True if the mechanism can be used with C_MessageEncryptInit
CKF_MESSAGE_DECRYPT	0x00000004	True if the mechanism can be used with C_MessageDecryptInit
CKF_MESSAGE_SIGN	0x00000008	True if the mechanism can be used with C_MessageSignInit
CKF_MESSAGE_VERIFY	0x00000010	True if the mechanism can be used with C_MessageVerifyInit
CKF_MULTI_MESSAGE	0x00000020	True if the mechanism can be used with C_*MessageBegin . One of CKF_MESSAGE_* flag must also be set.
CKF_FIND_OBJECTS	0x00000040	This flag can be passed in as a parameter to C_CancelSession to cancel an active object search operation. Any other use of this flag is outside the scope of this standard.

Bit Flag	Mask	Meaning
CKF_ENCRYPT	0x00000100	True if the mechanism can be used with C_EncryptInit
CKF_DECRYPT	0x00000200	True if the mechanism can be used with C_DecryptInit
CKF_DIGEST	0x00000400	True if the mechanism can be used with C_DigestInit
CKF_SIGN	0x00000800	True if the mechanism can be used with C_SignInit
CKF_SIGN_RECOVER	0x00001000	True if the mechanism can be used with C_SignRecoverInit
CKF_VERIFY	0x00002000	True if the mechanism can be used with C_VerifyInit
CKF_VERIFY_RECOVER	0x00004000	True if the mechanism can be used with C_VerifyRecoverInit
CKF_GENERATE	0x00008000	True if the mechanism can be used with C_GenerateKey
CKF_GENERATE_KEY_PAIR	0x00010000	True if the mechanism can be used with C_GenerateKeyPair
CKF_WRAP	0x00020000	True if the mechanism can be used with C_WrapKey
CKF_UNWRAP	0x00040000	True if the mechanism can be used with C_UnwrapKey
CKF_DERIVE	0x00080000	True if the mechanism can be used with C_DeriveKey
CKF_EXTENSION	0x80000000	True if there is an extension to the flags; false if no extensions. MUST be false for this version.

803 CK_MECHANISM_INFO_PTR is a pointer to a CK_MECHANISM_INFO.

804 3.6 Function types

805 Cryptoki represents information about functions with the following data types:

806 ♦ CK_RV

807 **CK_RV** is a value that identifies the return value of a Cryptoki function. It is defined as follows:

```
808 typedef CK_ULONG CK_RV;
809
```

810 Vendor defined values for this type may also be specified.

```
811 CKR_VENDOR_DEFINED
812
```

813 Section 5.1 defines the meaning of each **CK_RV** value. Return values **CKR_VENDOR_DEFINED** and
814 above are permanently reserved for token vendors. For interoperability, vendors should register their
815 return values through the PKCS process.

816 ♦ **CK_NOTIFY**

817 **CK_NOTIFY** is the type of a pointer to a function used by Cryptoki to perform notification callbacks. It is
818 defined as follows:

```
819 typedef CK_CALLBACK_FUNCTION(CK_RV, CK_NOTIFY) (  
820     CK_SESSION_HANDLE hSession,  
821     CK_NOTIFICATION event,  
822     CK_VOID_PTR pApplication  
823 );  
824
```

825 The arguments to a notification callback function have the following meanings:

826	<i>hSession</i>	The handle of the session performing the callback
827	<i>event</i>	The type of notification callback
828	<i>pApplication</i>	An application-defined value. This is the same value as was passed 829 to C_OpenSession to open the session performing the callback

830 ♦ **CK_C_XXX**

831 Cryptoki also defines an entire family of other function pointer types. For each function **C_XXX** in the
832 Cryptoki API (see Section 4.12 for detailed information about each of them), Cryptoki defines a type
833 **CK_C_XXX**, which is a pointer to a function with the same arguments and return value as **C_XXX** has.
834 An appropriately-set variable of type **CK_C_XXX** may be used by an application to call the Cryptoki
835 function **C_XXX**.

836 ♦ **CK_FUNCTION_LIST;** **CK_FUNCTION_LIST_PTR;** 837 **CK_FUNCTION_LIST_PTR_PTR**

838 **CK_FUNCTION_LIST** is a structure which contains a Cryptoki version and a function pointer to each
839 function in the Cryptoki API. It is defined as follows:

```
840 typedef struct CK_FUNCTION_LIST {  
841     CK_VERSION version;  
842     CK_C_Initialize C_Initialize;  
843     CK_C_Finalize C_Finalize;  
844     CK_C_GetInfo C_GetInfo;  
845     CK_C_GetFunctionList C_GetFunctionList;  
846     CK_C_GetSlotList C_GetSlotList;  
847     CK_C_GetSlotInfo C_GetSlotInfo;  
848     CK_C_GetTokenInfo C_GetTokenInfo;  
849     CK_C_GetMechanismList C_GetMechanismList;  
850     CK_C_GetMechanismInfo C_GetMechanismInfo;  
851     CK_C_InitToken C_InitToken;  
852     CK_C_InitPIN C_InitPIN;  
853     CK_C_SetPIN C_SetPIN;  
854     CK_C_OpenSession C_OpenSession;  
855     CK_C_CloseSession C_CloseSession;  
856     CK_C_CloseAllSessions C_CloseAllSessions;  
857     CK_C_GetSessionInfo C_GetSessionInfo;  
858  
859     CK_C_GetOperationState C_GetOperationState;  
860     CK_C_SetOperationState C_SetOperationState;  
861     CK_C_Login C_Login;  
862     CK_C_Logout C_Logout;  
863     CK_C_CreateObject C_CreateObject;  
864     CK_C_CopyObject C_CopyObject;  
865     CK_C_DestroyObject C_DestroyObject;
```

```

866 CK_C_GetObjectSize C_GetObjectSize;
867 CK_C_GetAttributeValue C_GetAttributeValue;
868 CK_C_SetAttributeValue C_SetAttributeValue;
869 CK_C_FindObjectsInit C_FindObjectsInit;
870 CK_C_FindObjects C_FindObjects;
871 CK_C_FindObjectsFinal C_FindObjectsFinal;
872 CK_C_EncryptInit C_EncryptInit;
873 CK_C_Encrypt C_Encrypt;
874 CK_C_EncryptUpdate C_EncryptUpdate;
875 CK_C_EncryptFinal C_EncryptFinal;
876 CK_C_MessageEncryptInit C_MessageEncryptInit;
877 CK_C_EncryptMessage C_EncryptMessage ;
878 CK_C_EncryptMessageBegin C_EncryptMessageBegin ;
879 CK_C_EncryptMessageNext C_EncryptMessageNext ;
880 CK_C_EncryptMessageFinal C_EncryptMessageFinal ;
881 CK_C_DecryptInit C_DecryptInit;
882 CK_C_Decrypt C_Decrypt;
883 CK_C_DecryptUpdate C_DecryptUpdate;
884 CK_C_DecryptFinal C_DecryptFinal;
885 CK_C_DigestInit C_DigestInit;
886 CK_C_Digest C_Digest;
887 CK_C_DigestUpdate C_DigestUpdate;
888 CK_C_DigestKey C_DigestKey;
889 CK_C_DigestFinal C_DigestFinal;
890 CK_C_SignInit C_SignInit;
891 CK_C_Sign C_Sign;
892 CK_C_SignUpdate C_SignUpdate;
893 CK_C_SignFinal C_SignFinal;
894 CK_C_SignRecoverInit C_SignRecoverInit;
895 CK_C_SignRecover C_SignRecover;
896 CK_C_VerifyInit C_VerifyInit;
897 CK_C_Verify C_Verify;
898 CK_C_VerifyUpdate C_VerifyUpdate;
899 CK_C_VerifyFinal C_VerifyFinal;
900 CK_C_VerifyRecoverInit C_VerifyRecoverInit;
901 CK_C_VerifyRecover C_VerifyRecover;
902 CK_C_DigestEncryptUpdate C_DigestEncryptUpdate;
903 CK_C_DecryptDigestUpdate C_DecryptDigestUpdate;
904 CK_C_SignEncryptUpdate C_SignEncryptUpdate;
905 CK_C_DecryptVerifyUpdate C_DecryptVerifyUpdate;
906 CK_C_GenerateKey C_GenerateKey;
907 CK_C_GenerateKeyPair C_GenerateKeyPair;
908 CK_C_WrapKey C_WrapKey;
909 CK_C_UnwrapKey C_UnwrapKey;
910 CK_C_DeriveKey C_DeriveKey;
911 CK_C_SeedRandom C_SeedRandom;
912 CK_C_GenerateRandom C_GenerateRandom;
913 CK_C_GetFunctionStatus C_GetFunctionStatus;
914 CK_C_CancelFunction C_CancelFunction;
915 CK_C_WaitForSlotEvent C_WaitForSlotEvent;
916 } CK_FUNCTION_LIST;
917

```

918 Each Cryptoki library has a static **CK_FUNCTION_LIST** structure, and a pointer to it (or to a copy of it
919 which is also owned by the library) may be obtained by the **C_GetFunctionList** function (see Section
920 5.2). The value that this pointer points to can be used by an application to quickly find out where the
921 executable code for each function in the Cryptoki API is located. Every function in the Cryptoki API
922 MUST have an entry point defined in the Cryptoki library's **CK_FUNCTION_LIST** structure. If a particular
923 function in the Cryptoki API is not supported by a library, then the function pointer for that function in the
924 library's **CK_FUNCTION_LIST** structure should point to a function stub which simply returns
925 CKR_FUNCTION_NOT_SUPPORTED.

926 In this structure 'version' is the cryptoki specification version number. The major and minor versions must
927 be set to 0x02 and 0x28 indicating a version 2.40 compatible structure. The updated function list table for
928 this version of the specification may be returned via **C_GetInterfaceList** or **C_GetInterface**.

929

930 An application may or may not be able to modify a Cryptoki library's static **CK_FUNCTION_LIST**
931 structure. Whether or not it can, it should never attempt to do so.

932 PKCS #11 modules must not add new functions at the end of the **CK_FUNCTION_LIST** that are not
933 contained within the defined structure. If a PKCS#11 module needs to define additional functions, they
934 should be placed within a vendor defined interface returned via **C_GetInterfaceList** or **C_GetInterface**.

935 **CK_FUNCTION_LIST_PTR** is a pointer to a **CK_FUNCTION_LIST**.

936 **CK_FUNCTION_LIST_PTR_PTR** is a pointer to a **CK_FUNCTION_LIST_PTR**.

937

938 ♦ **CK_FUNCTION_LIST_3_0; CK_FUNCTION_LIST_3_0_PTR;**
939 **CK_FUNCTION_LIST_3_0_PTR_PTR**

940 **CK_FUNCTION_LIST_3_0** is a structure which contains the same function pointers as in
941 **CK_FUNCTION_LIST** and additional functions added to the end of the structure that were defined in
942 Cryptoki version 3.0. It is defined as follows:

```
943 typedef struct CK_FUNCTION_LIST_3_0 {  
944     CK_VERSION version;  
945     CK_C_Initialize C_Initialize;  
946     CK_C_Finalize C_Finalize;  
947     CK_C_GetInfo C_GetInfo;  
948     CK_C_GetFunctionList C_GetFunctionList;  
949     CK_C_GetSlotList C_GetSlotList;  
950     CK_C_GetSlotInfo C_GetSlotInfo;  
951     CK_C_GetTokenInfo C_GetTokenInfo;  
952     CK_C_GetMechanismList C_GetMechanismList;  
953     CK_C_GetMechanismInfo C_GetMechanismInfo;  
954     CK_C_InitToken C_InitToken;  
955     CK_C_InitPIN C_InitPIN;  
956     CK_C_SetPIN C_SetPIN;  
957     CK_C_OpenSession C_OpenSession;  
958     CK_C_CloseSession C_CloseSession;  
959     CK_C_CloseAllSessions C_CloseAllSessions;  
960     CK_C_GetSessionInfo C_GetSessionInfo;  
961     CK_C_GetOperationState C_GetOperationState;  
962     CK_C_SetOperationState C_SetOperationState;  
963     CK_C_Login C_Login;  
964     CK_C_Logout C_Logout;  
965     CK_C_CreateObject C_CreateObject;  
966     CK_C_CopyObject C_CopyObject;  
967     CK_C_DestroyObject C_DestroyObject;  
968     CK_C_GetObjectSize C_GetObjectSize;  
969     CK_C_GetAttributeValue C_GetAttributeValue;  
970     CK_C_SetAttributeValue C_SetAttributeValue;  
971     CK_C_FindObjectsInit C_FindObjectsInit;  
972     CK_C_FindObjects C_FindObjects;  
973     CK_C_FindObjectsFinal C_FindObjectsFinal;  
974     CK_C_EncryptInit C_EncryptInit;  
975     CK_C_Encrypt C_Encrypt;  
976     CK_C_EncryptUpdate C_EncryptUpdate;  
977     CK_C_EncryptFinal C_EncryptFinal;  
978     CK_C_DecryptInit C_DecryptInit;  
979     CK_C_Decrypt C_Decrypt;  
980     CK_C_DecryptUpdate C_DecryptUpdate;  
981     CK_C_DecryptFinal C_DecryptFinal;
```

```

982 CK_C_DigestInit C_DigestInit;
983 CK_C_Digest C_Digest;
984 CK_C_DigestUpdate C_DigestUpdate;
985 CK_C_DigestKey C_DigestKey;
986 CK_C_DigestFinal C_DigestFinal;
987 CK_C_SignInit C_SignInit;
988 CK_C_Sign C_Sign;
989 CK_C_SignUpdate C_SignUpdate;
990 CK_C_SignFinal C_SignFinal;
991 CK_C_SignRecoverInit C_SignRecoverInit;
992 CK_C_SignRecover C_SignRecover;
993 CK_C_VerifyInit C_VerifyInit;
994 CK_C_Verify C_Verify;
995 CK_C_VerifyUpdate C_VerifyUpdate;
996 CK_C_VerifyFinal C_VerifyFinal;
997 CK_C_VerifyRecoverInit C_VerifyRecoverInit;
998 CK_C_VerifyRecover C_VerifyRecover;
999 CK_C_DigestEncryptUpdate C_DigestEncryptUpdate;
1000 CK_C_DecryptDigestUpdate C_DecryptDigestUpdate;
1001 CK_C_SignEncryptUpdate C_SignEncryptUpdate;
1002 CK_C_DecryptVerifyUpdate C_DecryptVerifyUpdate;
1003 CK_C_GenerateKey C_GenerateKey;
1004 CK_C_GenerateKeyPair C_GenerateKeyPair;
1005 CK_C_WrapKey C_WrapKey;
1006 CK_C_UnwrapKey C_UnwrapKey;
1007 CK_C_DeriveKey C_DeriveKey;
1008 CK_C_SeedRandom C_SeedRandom;
1009 CK_C_GenerateRandom C_GenerateRandom;
1010 CK_C_GetFunctionStatus C_GetFunctionStatus;
1011 CK_C_CancelFunction C_CancelFunction;
1012 CK_C_WaitForSlotEvent C_WaitForSlotEvent;
1013 CK_C_GetInterfaceList C_GetInterfaceList;
1014 CK_C_GetInterface C_GetInterface;
1015 CK_C_LoginUser C_LoginUser;
1016 CK_C_SessionCancel C_SessionCancel;
1017 CK_C_MessageEncryptInit C_MessageEncryptInit;
1018 CK_C_EncryptMessage C_EncryptMessage;
1019 CK_C_EncryptMessageBegin C_EncryptMessageBegin;
1020 CK_C_EncryptMessageNext C_EncryptMessageNext;
1021 CK_C_MessageEncryptFinal C_MessageEncryptFinal;
1022 CK_C_MessageDecryptInit C_MessageDecryptInit;
1023 CK_C_DecryptMessage C_DecryptMessage;
1024 CK_C_DecryptMessageBegin C_DecryptMessageBegin;
1025 CK_C_DecryptMessageNext C_DecryptMessageNext;
1026 CK_C_MessageDecryptFinal C_MessageDecryptFinal;
1027 CK_C_MessageSignInit C_MessageSignInit;
1028 CK_C_SignMessage C_SignMessage;
1029 CK_C_SignMessageBegin C_SignMessageBegin;
1030 CK_C_SignMessageNext C_SignMessageNext;
1031 CK_C_MessageSignFinal C_MessageSignFinal;
1032 CK_C_MessageVerifyInit C_MessageVerifyInit;
1033 CK_C_VerifyMessage C_VerifyMessage;
1034 CK_C_VerifyMessageBegin C_VerifyMessageBegin;
1035 CK_C_VerifyMessageNext C_VerifyMessageNext;
1036 CK_C_MessageVerifyFinal C_MessageVerifyFinal;
1037 } CK_FUNCTION_LIST_3_0;
1038

```

1039 For a general description of **CK_FUNCTION_LIST_3_0** see **CK_FUNCTION_LIST**.

1040 In this structure, *version* is the cryptoki specification version number. It should match the value of
1041 *cryptokiVersion* returned in the **CK_INFO** structure, but must be 3.0 at minimum.

1042 This function list may be returned via **C_GetInterfaceList** or **C_GetInterface**

1043 **CK_FUNCTION_LIST_3_0_PTR** is a pointer to a **CK_FUNCTION_LIST_3_0**.

CK_FUNCTION_LIST_3_0_PTR_PTR is a pointer to a **CK_FUNCTION_LIST_3_0_PTR**.

◆ **CK_INTERFACE; CK_INTERFACE_PTR; CK_INTERFACE_PTR_PTR**

CK_INTERFACE is a structure which contains an interface name with a function list and flag.
It is defined as follows:

```
typedef struct CK_INTERFACE {  
    CK_UTF8CHAR_PTR pInterfaceName;  
    CK_VOID_PTR      pFunctionList;  
    CK_FLAGS          flags;  
} CK_INTERFACE;
```

The fields of the structure have the following meanings:

<i>pInterfaceName</i>	the name of the interface
<i>pFunctionList</i>	the interface function list which must always begin with a CK_VERSION structure as the first field
<i>flags</i>	bit flags specifying interface capabilities

The interface name “PKCS 11” is reserved for use by interfaces defined within the cryptoki specification. Interfaces starting with the string: “Vendor ” are reserved for vendor use and will not otherwise be defined as interfaces in the PKCS #11 specification. Vendors should supply new functions with interface names of “Vendor {vendor name}”. For example “Vendor ACME Inc”.

The following table defines the flags field:

Table 9, *CK_INTERFACE* Flags

Bit Flag	Mask	Meaning
CKF_INTERFACE_FORK_SAFE	0x00000001	The returned interface will have fork tolerant semantics. When the application forks, each process will get its own copy of all session objects, session states, login states, and encryption states. Each process will also maintain access to token objects with their previously supplied handles.

CK_INTERFACE_PTR is a pointer to a **CK_INTERFACE**.

CK_INTERFACE_PTR_PTR is a pointer to a **CK_INTERFACE_PTR**.

3.7 Locking-related types

The types in this section are provided solely for applications which need to access Cryptoki from multiple threads simultaneously. *Applications which will not do this need not use any of these types.*

1072 ♦ CK_CREATEMUTEX

1073 **CK_CREATEMUTEX** is the type of a pointer to an application-supplied function which creates a new
1074 mutex object and returns a pointer to it. It is defined as follows:

```
1075 typedef CK_CALLBACK_FUNCTION(CK_RV, CK_CREATEMUTEX) (  
1076     CK_VOID_PTR_PTR ppMutex  
1077 );  
1078
```

1079 Calling a CK_CREATEMUTEX function returns the pointer to the new mutex object in the location pointed
1080 to by ppMutex. Such a function should return one of the following values:

```
1081 CKR_OK, CKR_GENERAL_ERROR  
1082 CKR_HOST_MEMORY
```

1083 ♦ CK_DESTROYMUTEX

1084 **CK_DESTROYMUTEX** is the type of a pointer to an application-supplied function which destroys an
1085 existing mutex object. It is defined as follows:

```
1086 typedef CK_CALLBACK_FUNCTION(CK_RV, CK_DESTROYMUTEX) (  
1087     CK_VOID_PTR pMutex  
1088 );  
1089
```

1090 The argument to a CK_DESTROYMUTEX function is a pointer to the mutex object to be destroyed. Such
1091 a function should return one of the following values:

```
1092 CKR_OK, CKR_GENERAL_ERROR  
1093 CKR_HOST_MEMORY  
1094 CKR_MUTEX_BAD
```

1095 ♦ CK_LOCKMUTEX and CK_UNLOCKMUTEX

1096 **CK_LOCKMUTEX** is the type of a pointer to an application-supplied function which locks an existing
1097 mutex object. **CK_UNLOCKMUTEX** is the type of a pointer to an application-supplied function which
1098 unlocks an existing mutex object. The proper behavior for these types of functions is as follows:

- 1099 • If a CK_LOCKMUTEX function is called on a mutex which is not locked, the calling thread obtains a
1100 lock on that mutex and returns.
- 1101 • If a CK_LOCKMUTEX function is called on a mutex which is locked by some thread other than the
1102 calling thread, the calling thread blocks and waits for that mutex to be unlocked.
- 1103 • If a CK_LOCKMUTEX function is called on a mutex which is locked by the calling thread, the
1104 behavior of the function call is undefined.
- 1105 • If a CK_UNLOCKMUTEX function is called on a mutex which is locked by the calling thread, that
1106 mutex is unlocked and the function call returns. Furthermore:
 - 1107 ○ If exactly one thread was blocking on that particular mutex, then that thread stops blocking,
1108 obtains a lock on that mutex, and its CK_LOCKMUTEX call returns.
 - 1109 ○ If more than one thread was blocking on that particular mutex, then exactly one of the
1110 blocking threads is selected somehow. That lucky thread stops blocking, obtains a lock on
1111 the mutex, and its CK_LOCKMUTEX call returns. All other threads blocking on that particular
1112 mutex continue to block.
- 1113 • If a CK_UNLOCKMUTEX function is called on a mutex which is not locked, then the function call
1114 returns the error code CKR_MUTEX_NOT_LOCKED.
- 1115 • If a CK_UNLOCKMUTEX function is called on a mutex which is locked by some thread other than the
1116 calling thread, the behavior of the function call is undefined.

1117 **CK_LOCKMUTEX** is defined as follows:

```
1118 typedef CK_CALLBACK_FUNCTION(CK_RV, CK_LOCKMUTEX) (  
1119     CK_VOID_PTR pMutex  
1120 );  
1121
```

1122 The argument to a CK_LOCKMUTEX function is a pointer to the mutex object to be locked. Such a
1123 function should return one of the following values:

```
1124 CKR_OK, CKR_GENERAL_ERROR  
1125 CKR_HOST_MEMORY,  
1126 CKR_MUTEX_BAD  
1127
```

1128 **CK_UNLOCKMUTEX** is defined as follows:

```
1129 typedef CK_CALLBACK_FUNCTION(CK_RV, CK_UNLOCKMUTEX) (  
1130     CK_VOID_PTR pMutex  
1131 );  
1132
```

1133 The argument to a CK_UNLOCKMUTEX function is a pointer to the mutex object to be unlocked. Such a
1134 function should return one of the following values:

```
1135 CKR_OK, CKR_GENERAL_ERROR  
1136 CKR_HOST_MEMORY  
1137 CKR_MUTEX_BAD  
1138 CKR_MUTEX_NOT_LOCKED
```

1139 ♦ **CK_C_INITIALIZE_ARGS; CK_C_INITIALIZE_ARGS_PTR**

1140 **CK_C_INITIALIZE_ARGS** is a structure containing the optional arguments for the **C_Initialize** function.
1141 For this version of Cryptoki, these optional arguments are all concerned with the way the library deals
1142 with threads. **CK_C_INITIALIZE_ARGS** is defined as follows:

```
1143 typedef struct CK_C_INITIALIZE_ARGS {  
1144     CK_CREATEMUTEX CreateMutex;  
1145     CK_DESTROYMUTEX DestroyMutex;  
1146     CK_LOCKMUTEX LockMutex;  
1147     CK_UNLOCKMUTEX UnlockMutex;  
1148     CK_FLAGS flags;  
1149     CK_VOID_PTR pReserved;  
1150 } CK_C_INITIALIZE_ARGS;  
1151
```

1152 The fields of the structure have the following meanings:

1153	<i>CreateMutex</i>	pointer to a function to use for creating mutex objects
1154	<i>DestroyMutex</i>	pointer to a function to use for destroying mutex objects
1155	<i>LockMutex</i>	pointer to a function to use for locking mutex objects
1156	<i>UnlockMutex</i>	pointer to a function to use for unlocking mutex objects
1157	<i>flags</i>	bit flags specifying options for C_Initialize ; the flags are defined below
1158		
1159	<i>pReserved</i>	reserved for future use. Should be NULL_PTR for this version of Cryptoki
1160		

1161 The following table defines the flags field:

1162 *Table 10, C_Initialize Parameter Flags*

Bit Flag	Mask	Meaning
CKF_LIBRARY_CANT_CREATE_OS_THREADS	0x00000001	True if application threads which are executing calls to the library may <i>not</i> use native operating system calls to spawn new threads; false if they may
CKF_OS_LOCKING_OK	0x00000002	True if the library can use the native operation system threading model for locking; false otherwise

1163 CK_C_INITIALIZE_ARGS_PTR is a pointer to a CK_C_INITIALIZE_ARGS.

4 Objects

Cryptoki recognizes a number of classes of objects, as defined in the **CK_OBJECT_CLASS** data type. An object consists of a set of attributes, each of which has a given value. Each attribute that an object possesses has precisely one value. The following figure illustrates the high-level hierarchy of the Cryptoki objects and some of the attributes they support:

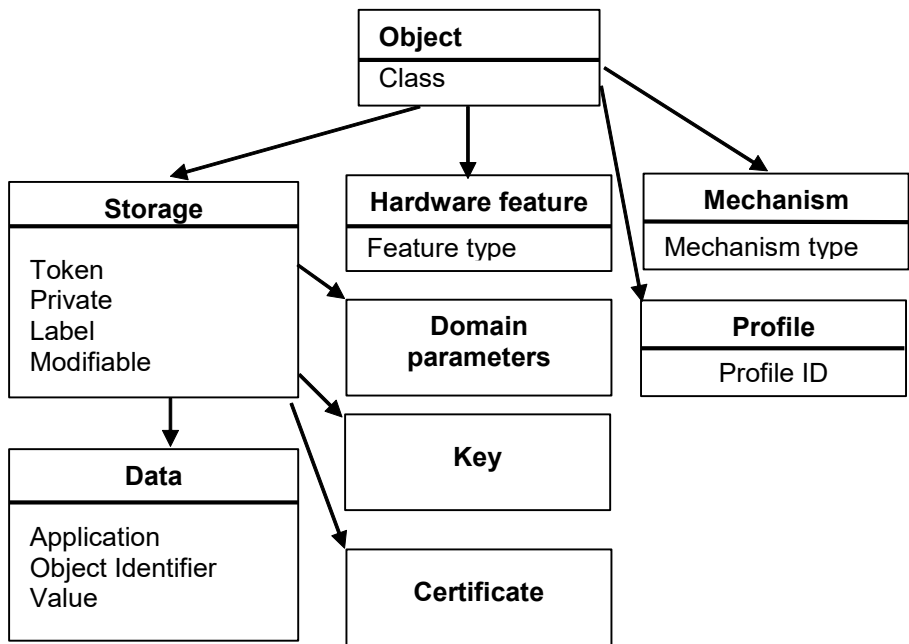


Figure 1, Object Attribute Hierarchy

Cryptoki provides functions for creating, destroying, and copying objects in general, and for obtaining and modifying the values of their attributes. Some of the cryptographic functions (e.g., **C_GenerateKey**) also create key objects to hold their results.

Objects are always “well-formed” in Cryptoki—that is, an object always contains all required attributes, and the attributes are always consistent with one another from the time the object is created. This contrasts with some object-based paradigms where an object has no attributes other than perhaps a class when it is created, and is uninitialized for some time. In Cryptoki, objects are always initialized.

Tables throughout most of Section 4 define each Cryptoki attribute in terms of the data type of the attribute value and the meaning of the attribute, which may include a default initial value. Some of the data types are defined explicitly by Cryptoki (e.g., **CK_OBJECT_CLASS**). Attribute values may also take the following types:

Byte array	an arbitrary string (array) of CK_BYTES
Big integer	a string of CK_BYTES representing an unsigned integer of arbitrary size, most-significant byte first (e.g., the integer 32768 is represented as the 2-byte string 0x80 0x00)
Local string	an unpadded string of CK_CHARS (see Table 3) with no null-termination
RFC2279 string	an unpadded string of CK_UTF8CHARs with no null-termination

A token can hold several identical objects, *i.e.*, it is permissible for two or more objects to have exactly the same values for all their attributes.

In most cases each type of object in the Cryptoki specification possesses a completely well-defined set of Cryptoki attributes. Some of these attributes possess default values, and need not be specified when creating an object; some of these default values may even be the empty string (""). Nonetheless, the object possesses these attributes. A given object has a single value for each attribute it possesses, even if the attribute is a vendor-specific attribute whose meaning is outside the scope of Cryptoki.

In addition to possessing Cryptoki attributes, objects may possess additional vendor-specific attributes whose meanings and values are not specified by Cryptoki.

4.1 Creating, modifying, and copying objects

All Cryptoki functions that create, modify, or copy objects take a template as one of their arguments, where the template specifies attribute values. Cryptographic functions that create objects (see Section 5.18) may also contribute some additional attribute values themselves; which attributes have values contributed by a cryptographic function call depends on which cryptographic mechanism is being performed (see [PKCS11-Curr] and [PKCS11-Hist] for specification of mechanisms for PKCS #11). In any case, all the required attributes supported by an object class that do not have default values **MUST** be specified when an object is created, either in the template or by the function itself.

4.1.1 Creating objects

Objects may be created with the Cryptoki functions **C_CreateObject** (see Section 5.7), **C_GenerateKey**, **C_GenerateKeyPair**, **C_UnwrapKey**, and **C_DeriveKey** (see Section 5.18). In addition, copying an existing object (with the function **C_CopyObject**) also creates a new object, but we consider this type of object creation separately in Section 4.1.3.

Attempting to create an object with any of these functions requires an appropriate template to be supplied.

1. If the supplied template specifies a value for an invalid attribute, then the attempt should fail with the error code **CKR_ATTRIBUTE_TYPE_INVALID**. An attribute is valid if it is either one of the attributes described in the Cryptoki specification or an additional vendor-specific attribute supported by the library and token.
2. If the supplied template specifies an invalid value for a valid attribute, then the attempt should fail with the error code **CKR_ATTRIBUTE_VALUE_INVALID**. The valid values for Cryptoki attributes are described in the Cryptoki specification.
3. If the supplied template specifies a value for a read-only attribute, then the attempt should fail with the error code **CKR_ATTRIBUTE_READ_ONLY**. Whether or not a given Cryptoki attribute is read-only is explicitly stated in the Cryptoki specification; however, a particular library and token may be even more restrictive than Cryptoki specifies. In other words, an attribute which Cryptoki says is not read-only may nonetheless be read-only under certain circumstances (*i.e.*, in conjunction with some combinations of other attributes) for a particular library and token. Whether or not a given non-Cryptoki attribute is read-only is obviously outside the scope of Cryptoki.
4. If the attribute values in the supplied template, together with any default attribute values and any attribute values contributed to the object by the object-creation function itself, are insufficient to fully specify the object to create, then the attempt should fail with the error code **CKR_TEMPLATE_INCOMPLETE**.
5. If the attribute values in the supplied template, together with any default attribute values and any attribute values contributed to the object by the object-creation function itself, are inconsistent, then the attempt should fail with the error code **CKR_TEMPLATE_INCONSISTENT**. A set of attribute values is inconsistent if not all of its members can be satisfied simultaneously *by the token*, although each value individually is valid in Cryptoki. One example of an inconsistent template would be using a template

which specifies two different values for the same attribute. Another example would be trying to create a secret key object with an attribute which is appropriate for various types of public keys or private keys, but not for secret keys. A final example would be a template with an attribute that violates some token specific requirement. Note that this final example of an inconsistent template is token-dependent—on a different token, such a template might *not* be inconsistent.

6. If the supplied template specifies the same value for a particular attribute more than once (or the template specifies the same value for a particular attribute that the object-creation function itself contributes to the object), then the behavior of Cryptoki is not completely specified. The attempt to create an object can either succeed—thereby creating the same object that would have been created if the multiply-specified attribute had only appeared once—or it can fail with error code CKR_TEMPLATE_INCONSISTENT. Library developers are encouraged to make their libraries behave as though the attribute had only appeared once in the template; application developers are strongly encouraged never to put a particular attribute into a particular template more than once.

If more than one of the situations listed above applies to an attempt to create an object, then the error code returned from the attempt can be any of the error codes from above that applies.

4.1.2 Modifying objects

Objects may be modified with the Cryptoki function **C_SetAttributeValue** (see Section 5.7). The template supplied to **C_SetAttributeValue** can contain new values for attributes which the object already possesses; values for attributes which the object does not yet possess; or both.

Some attributes of an object may be modified after the object has been created, and some may not. In addition, attributes which Cryptoki specifies are modifiable may actually *not* be modifiable on some tokens. That is, if a Cryptoki attribute is described as being modifiable, that really means only that it is modifiable *insofar as the Cryptoki specification is concerned*. A particular token might not actually support modification of some such attributes. Furthermore, whether or not a particular attribute of an object on a particular token is modifiable might depend on the values of certain attributes of the object. For example, a secret key object's **CKA_SENSITIVE** attribute can be changed from CK_FALSE to CK_TRUE, but not the other way around.

All the scenarios in Section 4.1.1—and the error codes they return—apply to modifying objects with **C_SetAttributeValue**, except for the possibility of a template being incomplete.

4.1.3 Copying objects

Unless an object's CKA_COPYABLE (see table 21) attribute is set to CK_FALSE, it may be copied with the Cryptoki function **C_CopyObject** (see Section 5.7). In the process of copying an object, **C_CopyObject** also modifies the attributes of the newly-created copy according to an application-supplied template.

The Cryptoki attributes which can be modified during the course of a **C_CopyObject** operation are the same as the Cryptoki attributes which are described as being modifiable, plus the four special attributes **CKA_TOKEN**, **CKA_PRIVATE**, **CKA_MODIFIABLE** and **CKA_DESTROYABLE**. To be more precise, these attributes are modifiable during the course of a **C_CopyObject** operation *insofar as the Cryptoki specification is concerned*. A particular token might not actually support modification of some such attributes during the course of a **C_CopyObject** operation. Furthermore, whether or not a particular attribute of an object on a particular token is modifiable during the course of a **C_CopyObject** operation might depend on the values of certain attributes of the object. For example, a secret key object's **CKA_SENSITIVE** attribute can be changed from CK_FALSE to CK_TRUE during the course of a **C_CopyObject** operation, but not the other way around.

If the CKA_COPYABLE attribute of the object to be copied is set to CK_FALSE, **C_CopyObject** returns CKR_ACTION_PROHIBITED. Otherwise, the scenarios described in 10.1.1 - and the error codes they return - apply to copying objects with **C_CopyObject**, except for the possibility of a template being incomplete.

4.2 Common attributes

Table 11, Common footnotes for object attribute tables

- ¹ MUST be specified when object is created with **C_CreateObject**.
- ² MUST *not* be specified when object is created with **C_CreateObject**.
- ³ MUST be specified when object is generated with **C_GenerateKey** or **C_GenerateKeyPair**.
- ⁴ MUST *not* be specified when object is generated with **C_GenerateKey** or **C_GenerateKeyPair**.
- ⁵ MUST be specified when object is unwrapped with **C_UnwrapKey**.
- ⁶ MUST *not* be specified when object is unwrapped with **C_UnwrapKey**.
- ⁷ Cannot be revealed if object has its **CKA_SENSITIVE** attribute set to CK_TRUE or its **CKA_EXTRACTABLE** attribute set to CK_FALSE.
- ⁸ May be modified after object is created with a **C_SetAttributeValue** call, or in the process of copying object with a **C_CopyObject** call. However, it is possible that a particular token may not permit modification of the attribute during the course of a **C_CopyObject** call.
- ⁹ Default value is token-specific, and may depend on the values of other attributes.
- ¹⁰ Can only be set to CK_TRUE by the SO user.
- ¹¹ Attribute cannot be changed once set to CK_TRUE. It becomes a read only attribute.
- ¹² Attribute cannot be changed once set to CK_FALSE. It becomes a read only attribute.

Table 12, Common Object Attributes

Attribute	Data Type	Meaning
CKA_CLASS ¹	CK_OBJECT_CLASS	Object class (type)

Refer to Table 11 for footnotes

The above table defines the attributes common to all objects.

4.3 Hardware Feature Objects

4.3.1 Definitions

This section defines the object class CKO_HW_FEATURE for type CK_OBJECT_CLASS as used in the CKA_CLASS attribute of objects.

4.3.2 Overview

Hardware feature objects (**CKO_HW_FEATURE**) represent features of the device. They provide an easily expandable method for introducing new value-based features to the Cryptoki interface.

When searching for objects using **C_FindObjectsInit** and **C_FindObjects**, hardware feature objects are not returned unless the **CKA_CLASS** attribute in the template has the value **CKO_HW_FEATURE**. This protects applications written to previous versions of Cryptoki from finding objects that they do not understand.

Table 13, Hardware Feature Common Attributes

Attribute	Data Type	Meaning
CKA_HW_FEATURE_TYPE ¹	CK_HW_FEATURE_TYPE	Hardware feature (type)

Refer to Table 11 for footnotes

4.3.3 Clock

4.3.3.1 Definition

The CKA_HW_FEATURE_TYPE attribute takes the value CKH_CLOCK of type CK_HW_FEATURE_TYPE.

4.3.3.2 Description

Clock objects represent real-time clocks that exist on the device. This represents the same clock source as the **utcTime** field in the **CK_TOKEN_INFO** structure.

Table 14, Clock Object Attributes

Attribute	Data Type	Meaning
CKA_VALUE	CK_CHAR[16]	Current time as a character-string of length 16, represented in the format YYYYMMDDhhmmssxx (4 characters for the year; 2 characters each for the month, the day, the hour, the minute, and the second; and 2 additional reserved '0' characters).

The **CKA_VALUE** attribute may be set using the **C_SetAttributeValue** function if permitted by the device. The session used to set the time MUST be logged in. The device may require the SO to be the user logged in to modify the time value. **C_SetAttributeValue** will return the error CKR_USER_NOT_LOGGED_IN to indicate that a different user type is required to set the value.

4.3.4 Monotonic Counter Objects

4.3.4.1 Definition

The CKA_HW_FEATURE_TYPE attribute takes the value CKH_MONOTONIC_COUNTER of type CK_HW_FEATURE_TYPE.

4.3.4.2 Description

Monotonic counter objects represent hardware counters that exist on the device. The counter is guaranteed to increase each time its value is read, but not necessarily by one. This might be used by an application for generating serial numbers to get some assurance of uniqueness per token.

Table 15, Monotonic Counter Attributes

Attribute	Data Type	Meaning
CKA_RESET_ON_INIT ¹	CK_BBOOL	The value of the counter will reset to a previously returned value if the token is initialized using C_InitToken .
CKA_HAS_RESET ¹	CK_BBOOL	The value of the counter has been reset at least once at some point in time.
CKA_VALUE ¹	Byte Array	The current version of the monotonic counter. The value is returned in big endian order.

¹Read Only

The **CKA_VALUE** attribute may not be set by the client.

4.3.5 User Interface Objects

4.3.5.1 Definition

The CKA_HW_FEATURE_TYPE attribute takes the value CKH_USER_INTERFACE of type CK_HW_FEATURE_TYPE.

4.3.5.2 Description

User interface objects represent the presentation capabilities of the device.

Table 16, User Interface Object Attributes

Attribute	Data type	Meaning
CKA_PIXEL_X	CK_ULONG	Screen resolution (in pixels) in X-axis (e.g. 1280)
CKA_PIXEL_Y	CK_ULONG	Screen resolution (in pixels) in Y-axis (e.g. 1024)
CKA_RESOLUTION	CK_ULONG	DPI, pixels per inch
CKA_CHAR_ROWS	CK_ULONG	For character-oriented displays; number of character rows (e.g. 24)
CKA_CHAR_COLUMNS	CK_ULONG	For character-oriented displays: number of character columns (e.g. 80). If display is of proportional-font type, this is the width of the display in "em"-s (letter "M"), see CC/PP Struct.
CKA_COLOR	CK_BBOOL	Color support
CKA_BITS_PER_PIXEL	CK_ULONG	The number of bits of color or grayscale information per pixel.
CKA_CHAR_SETS	RFC 2279 string	String indicating supported character sets, as defined by IANA MIBenum sets (www.iana.org). Supported character sets are separated with ";". E.g. a token supporting iso-8859-1 and US-ASCII would set the attribute value to "4;3".
CKA_ENCODING_METHODS	RFC 2279 string	String indicating supported content transfer encoding methods, as defined by IANA (www.iana.org). Supported methods are separated with ";". E.g. a token supporting 7bit, 8bit and base64 could set the attribute value to "7bit;8bit;base64".
CKA_MIME_TYPES	RFC 2279 string	String indicating supported (presentable) MIME-types, as defined by IANA (www.iana.org). Supported types are separated with ";". E.g. a token supporting MIME types "a/b", "a/c" and "a/d" would set the attribute value to "a/b;a/c;a/d".

The selection of attributes, and associated data types, has been done in an attempt to stay as aligned with RFC 2534 and CC/PP Struct as possible. The special value CK_UNAVAILABLE_INFORMATION may be used for CK_ULONG-based attributes when information is not available or applicable.

None of the attribute values may be set by an application.

The value of the **CKA_ENCODING_METHODS** attribute may be used when the application needs to send MIME objects with encoded content to the token.

4.4 Storage Objects

This is not an object class; hence no CKO_ definition is required. It is a category of object classes with common attributes for the object classes that follow.

Table 17, Common Storage Object Attributes

Attribute	Data Type	Meaning
CKA_TOKEN	CK_BBOOL	CK_TRUE if object is a token object; CK_FALSE if object is a session object. Default is CK_FALSE.
CKA_PRIVATE	CK_BBOOL	CK_TRUE if object is a private object; CK_FALSE if object is a public object. Default value is token-specific, and may depend on the values of other attributes of the object.
CKA_MODIFIABLE	CK_BBOOL	CK_TRUE if object can be modified. Default is CK_TRUE.
CKA_LABEL	RFC2279 string	Description of the object (default empty).
CKA_COPYABLE	CK_BBOOL	CK_TRUE if object can be copied using C_CopyObject. Defaults to CK_TRUE. Can't be set to TRUE once it is set to FALSE.
CKA_DESTROYABLE	CK_BBOOL	CK_TRUE if the object can be destroyed using C_DestroyObject. Default is CK_TRUE.
CKA_UNIQUE_ID ²⁴⁶	RFC2279 string	The unique identifier assigned to the object.

Only the **CKA_LABEL** attribute can be modified after the object is created. (The **CKA_TOKEN**, **CKA_PRIVATE**, and **CKA_MODIFIABLE** attributes can be changed in the process of copying an object, however.)

The **CKA_TOKEN** attribute identifies whether the object is a token object or a session object.

When the **CKA_PRIVATE** attribute is CK_TRUE, a user may not access the object until the user has been authenticated to the token.

The value of the **CKA_MODIFIABLE** attribute determines whether or not an object is read-only.

The **CKA_LABEL** attribute is intended to assist users in browsing.

The value of the CKA_COPYABLE attribute determines whether or not an object can be copied. This attribute can be used in conjunction with CKA_MODIFIABLE to prevent changes to the permitted usages of keys and other objects.

The value of the CKA_DESTROYABLE attribute determines whether the object can be destroyed using C_DestroyObject.

4.4.1 The CKA_UNIQUE_ID attribute

Any time a new object is created, a value for CKA_UNIQUE_ID MUST be generated by the token and stored with the object. The specific algorithm used to generate unique ID values for objects is token-specific, but values generated MUST be unique across all objects visible to any particular session, and SHOULD be unique across all objects created by the token. Reinitializing the token, such as by calling C_InitToken, MAY cause reuse of CKA_UNIQUE_ID values.

Any attempt to modify the CKA_UNIQUE_ID attribute of an existing object or to specify the value of the CKA_UNIQUE_ID attribute in the template for an operation that creates one or more objects MUST fail. Operations failing for this reason return the error code CKR_ATTRIBUTE_READ_ONLY.

4.5 Data objects

4.5.1 Definitions

This section defines the object class CKO_DATA for type CK_OBJECT_CLASS as used in the CKA_CLASS attribute of objects.

4.5.2 Overview

Data objects (object class **CKO_DATA**) hold information defined by an application. Other than providing access to it, Cryptoki does not attach any special meaning to a data object. The following table lists the attributes supported by data objects, in addition to the common attributes defined for this object class:

Table 18, Data Object Attributes

Attribute	Data type	Meaning
CKA_APPLICATION	RFC2279 string	Description of the application that manages the object (default empty)
CKA_OBJECT_ID	Byte Array	DER-encoding of the object identifier indicating the data object type (default empty)
CKA_VALUE	Byte array	Value of the object (default empty)

The **CKA_APPLICATION** attribute provides a means for applications to indicate ownership of the data objects they manage. Cryptoki does not provide a means of ensuring that only a particular application has access to a data object, however.

The **CKA_OBJECT_ID** attribute provides an application independent and expandable way to indicate the type of the data object value. Cryptoki does not provide a means of insuring that the data object identifier matches the data value.

The following is a sample template containing attributes for creating a data object:

```
CK_OBJECT_CLASS class = CKO_DATA;
CK_UTF8CHAR label[] = "A data object";
CK_UTF8CHAR application[] = "An application";
CK_BYTE data[] = "Sample data";
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_APPLICATION, application, sizeof(application)-1},
    {CKA_VALUE, data, sizeof(data)}
};
```

4.6 Certificate objects

4.6.1 Definitions

This section defines the object class CKO_CERTIFICATE for type CK_OBJECT_CLASS as used in the CKA_CLASS attribute of objects.

4.6.2 Overview

Certificate objects (object class **CKO_CERTIFICATE**) hold public-key or attribute certificates. Other than providing access to certificate objects, Cryptoki does not attach any special meaning to certificates. The following table defines the common certificate object attributes, in addition to the common attributes defined for this object class:

Table 19, Common Certificate Object Attributes

Attribute	Data type	Meaning
CKA_CERTIFICATE_TYPE ¹	CK_CERTIFICATE_TYPE	Type of certificate
CKA_TRUSTED ¹⁰	CK_BBOOL	The certificate can be trusted for the application that it was created.
CKA_CERTIFICATE_CATEGORY	CKA_CERTIFICATE_CATEGORY	(default CK_CERTIFICATE_CATEGORY_UNSPECIFIED)
CKA_CHECK_VALUE	Byte array	Checksum
CKA_START_DATE	CK_DATE	Start date for the certificate (default empty)
CKA_END_DATE	CK_DATE	End date for the certificate (default empty)
CKA_PUBLIC_KEY_INFO	Byte Array	DER-encoding of the SubjectPublicKeyInfo for the public key contained in this certificate (default empty)

1404 ¹ Refer to Table 11 for footnotes

1405 Cryptoki does not enforce the relationship of the CKA_PUBLIC_KEY_INFO to the public key in the
1406 certificate, but does recommend that the key be extracted from the certificate to create this value.

1407 The **CKA_CERTIFICATE_TYPE** attribute may not be modified after an object is created. This version of
1408 Cryptoki supports the following certificate types:

- 1409 • X.509 public key certificate
- 1410 • WTLS public key certificate
- 1411 • X.509 attribute certificate

1412 The **CKA_TRUSTED** attribute cannot be set to CK_TRUE by an application. It MUST be set by a token
1413 initialization application or by the token's SO. Trusted certificates cannot be modified.

1414 The **CKA_CERTIFICATE_CATEGORY** attribute is used to indicate if a stored certificate is a user
1415 certificate for which the corresponding private key is available on the token ("token user"), a CA certificate
1416 ("authority"), or another end-entity certificate ("other entity"). This attribute may not be modified after an
1417 object is created.

1418 The **CKA_CERTIFICATE_CATEGORY** and **CKA_TRUSTED** attributes will together be used to map to
1419 the categorization of the certificates.

1420 **CKA_CHECK_VALUE**: The value of this attribute is derived from the certificate by taking the first three
1421 bytes of the SHA-1 hash of the certificate object's CKA_VALUE attribute.

1422 The **CKA_START_DATE** and **CKA_END_DATE** attributes are for reference only; Cryptoki does not
1423 attach any special meaning to them. When present, the application is responsible to set them to values
1424 that match the certificate's encoded "not before" and "not after" fields (if any).

1425 4.6.3 X.509 public key certificate objects

1426 X.509 certificate objects (certificate type **CKC_X_509**) hold X.509 public key certificates. The following
1427 table defines the X.509 certificate object attributes, in addition to the common attributes defined for this
1428 object class:

Attribute	Data type	Meaning
CKA_SUBJECT ¹	Byte array	DER-encoding of the certificate subject name
CKA_ID	Byte array	Key identifier for public/private key pair (default empty)
CKA_ISSUER	Byte array	DER-encoding of the certificate issuer name (default empty)
CKA_SERIAL_NUMBER	Byte array	DER-encoding of the certificate serial number (default empty)
CKA_VALUE ²	Byte array	BER-encoding of the certificate
CKA_URL ³	RFC2279 string	If not empty this attribute gives the URL where the complete certificate can be obtained (default empty)
CKA_HASH_OF_SUBJECT_PUBLIC_KEY ⁴	Byte array	Hash of the subject public key (default empty). Hash algorithm is defined by CKA_NAME_HASH_ALGORITHM
CKA_HASH_OF_ISSUER_PUBLIC_KEY ⁴	Byte array	Hash of the issuer public key (default empty). Hash algorithm is defined by CKA_NAME_HASH_ALGORITHM
CKA_JAVA_MIDP_SECURITY_DOMAIN	CK_JAVA_MIDP_SECURITY_DOMAIN	Java MIDP security domain. (default CK_SECURITY_DOMAIN_UNSPECIFIED)
CKA_NAME_HASH_ALGORITHM	CK_MECHANISM_TYPE	Defines the mechanism used to calculate CKA_HASH_OF_SUBJECT_PUBLIC_KEY and CKA_HASH_OF_ISSUER_PUBLIC_KEY. If the attribute is not present then the type defaults to SHA-1.

1430 ¹MUST be specified when the object is created.

1431 ²MUST be specified when the object is created. MUST be non-empty if CKA_URL is empty.

1432 ³MUST be non-empty if CKA_VALUE is empty.

1433 ⁴Can only be empty if CKA_URL is empty.

1434 Only the **CKA_ID**, **CKA_ISSUER**, and **CKA_SERIAL_NUMBER** attributes may be modified after the
1435 object is created.

1436 The **CKA_ID** attribute is intended as a means of distinguishing multiple public-key/private-key pairs held
1437 by the same subject (whether stored in the same token or not). (Since the keys are distinguished by
1438 subject name as well as identifier, it is possible that keys for different subjects may have the same
1439 **CKA_ID** value without introducing any ambiguity.)

1440 It is intended in the interests of interoperability that the subject name and key identifier for a certificate will
1441 be the same as those for the corresponding public and private keys (though it is not required that all be
1442 stored in the same token). However, Cryptoki does not enforce this association, or even the uniqueness
1443 of the key identifier for a given subject; in particular, an application may leave the key identifier empty.

1444 The **CKA_ISSUER** and **CKA_SERIAL_NUMBER** attributes are for compatibility with PKCS #7 and
1445 Privacy Enhanced Mail (RFC1421). Note that with the version 3 extensions to X.509 certificates, the key
1446 identifier may be carried in the certificate. It is intended that the **CKA_ID** value be identical to the key
1447 identifier in such a certificate extension, although this will not be enforced by Cryptoki.

The **CKA_URL** attribute enables the support for storage of the URL where the certificate can be found instead of the certificate itself. Storage of a URL instead of the complete certificate is often used in mobile environments.

The **CKA_HASH_OF_SUBJECT_PUBLIC_KEY** and **CKA_HASH_OF_ISSUER_PUBLIC_KEY** attributes are used to store the hashes of the public keys of the subject and the issuer. They are particularly important when only the URL is available to be able to correlate a certificate with a private key and when searching for the certificate of the issuer. The hash algorithm is defined by **CKA_NAME_HASH_ALGORITHM**.

The **CKA_JAVA_MIDP_SECURITY_DOMAIN** attribute associates a certificate with a Java MIDP security domain.

The following is a sample template for creating an X.509 certificate object:

```
CK_OBJECT_CLASS class = CKO_CERTIFICATE;
CK_CERTIFICATE_TYPE certType = CKC_X_509;
CK_UTF8CHAR label[] = "A certificate object";
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BYTE certificate[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_CERTIFICATE_TYPE, &certType, sizeof(certType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_SUBJECT, subject, sizeof(subject)},
    {CKA_ID, id, sizeof(id)},
    {CKA_VALUE, certificate, sizeof(certificate)}
};
```

4.6.4 WTLS public key certificate objects

WTLS certificate objects (certificate type **CKC_WTLS**) hold WTLS public key certificates. The following table defines the WTLS certificate object attributes, in addition to the common attributes defined for this object class.

Table 21: WTLS Certificate Object Attributes

Attribute	Data type	Meaning
CKA_SUBJECT ¹	Byte array	WTLS-encoding (Identifier type) of the certificate subject
CKA_ISSUER	Byte array	WTLS-encoding (Identifier type) of the certificate issuer (default empty)
CKA_VALUE ²	Byte array	WTLS-encoding of the certificate
CKA_URL ³	RFC2279 string	If not empty this attribute gives the URL where the complete certificate can be obtained
CKA_HASH_OF_SUBJECT_PUBLIC_KEY ⁴	Byte array	SHA-1 hash of the subject public key (default empty). Hash algorithm is defined by CKA_NAME_HASH_ALGORITHM
CKA_HASH_OF_ISSUER_PUBLIC_KEY ⁴	Byte array	SHA-1 hash of the issuer public key (default empty). Hash algorithm is defined by CKA_NAME_HASH_ALGORITHM
CKA_NAME_HASH_ALGORITHM	CK_MECHANISM_TYPE	Defines the mechanism used to calculate CKA_HASH_OF_SUBJECT_PUBLIC

Attribute	Data type	Meaning
		_KEY and CKA_HASH_OF_ISSUER_PUBLIC_KEY. If the attribute is not present then the type defaults to SHA-1.

¹MUST be specified when the object is created. Can only be empty if CKA_VALUE is empty.

²MUST be specified when the object is created. MUST be non-empty if CKA_URL is empty.

³MUST be non-empty if CKA_VALUE is empty.

⁴Can only be empty if CKA_URL is empty.

Only the **CKA_ISSUER** attribute may be modified after the object has been created.

The encoding for the **CKA_SUBJECT**, **CKA_ISSUER**, and **CKA_VALUE** attributes can be found in [WTLS].

The **CKA_URL** attribute enables the support for storage of the URL where the certificate can be found instead of the certificate itself. Storage of a URL instead of the complete certificate is often used in mobile environments.

The **CKA_HASH_OF_SUBJECT_PUBLIC_KEY** and **CKA_HASH_OF_ISSUER_PUBLIC_KEY** attributes are used to store the hashes of the public keys of the subject and the issuer. They are particularly important when only the URL is available to be able to correlate a certificate with a private key and when searching for the certificate of the issuer. The hash algorithm is defined by CKA_NAME_HASH_ALGORITHM.

The following is a sample template for creating a WTLS certificate object:

```
CK_OBJECT_CLASS class = CKO_CERTIFICATE;
CK_CERTIFICATE_TYPE certType = CKC_WTLS;
CK_UTF8CHAR label[] = "A certificate object";
CK_BYTE subject[] = {...};
CK_BYTE certificate[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] =
{
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_CERTIFICATE_TYPE, &certType, sizeof(certType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_SUBJECT, subject, sizeof(subject)},
    {CKA_VALUE, certificate, sizeof(certificate)}
};
```

4.6.5 X.509 attribute certificate objects

X.509 attribute certificate objects (certificate type **CKC_X_509_ATTR_CERT**) hold X.509 attribute certificates. The following table defines the X.509 attribute certificate object attributes, in addition to the common attributes defined for this object class:

Table 22, X.509 Attribute Certificate Object Attributes

Attribute	Data Type	Meaning
CKA_OWNER ¹	Byte Array	DER-encoding of the attribute certificate's subject field. This is distinct from the CKA_SUBJECT attribute contained in CKC_X_509 certificates because the ASN.1 syntax and encoding are different.
CKA_AC_ISSUER	Byte Array	DER-encoding of the attribute certificate's issuer field. This is distinct from the CKA_ISSUER attribute contained in CKC_X_509 certificates because the ASN.1 syntax and encoding are different. (default empty)
CKA_SERIAL_NUMBER	Byte Array	DER-encoding of the certificate serial number. (default empty)
CKA_ATTR_TYPES	Byte Array	BER-encoding of a sequence of object identifier values corresponding to the attribute types contained in the certificate. When present, this field offers an opportunity for applications to search for a particular attribute certificate without fetching and parsing the certificate itself. (default empty)
CKA_VALUE ¹	Byte Array	BER-encoding of the certificate.

¹MUST be specified when the object is created

Only the **CKA_AC_ISSUER**, **CKA_SERIAL_NUMBER** and **CKA_ATTR_TYPES** attributes may be modified after the object is created.

The following is a sample template for creating an X.509 attribute certificate object:

```

CK_OBJECT_CLASS class = CKO_CERTIFICATE;
CK_CERTIFICATE_TYPE certType = CKC_X_509_ATTR_CERT;
CK_UTF8CHAR label[] = "An attribute certificate object";
CK_BYTE owner[] = {...};
CK_BYTE certificate[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_CERTIFICATE_TYPE, &certType, sizeof(certType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_OWNER, owner, sizeof(owner)},
    {CKA_VALUE, certificate, sizeof(certificate)}
};

```

4.7 Key objects

4.7.1 Definitions

There is no CKO_ definition for the base key object class, only for the key types derived from it.

This section defines the object class CKO_PUBLIC_KEY, CKO_PRIVATE_KEY and CKO_SECRET_KEY for type CK_OBJECT_CLASS as used in the CKA_CLASS attribute of objects.

4.7.2 Overview

Key objects hold encryption or authentication keys, which can be public keys, private keys, or secret keys. The following common footnotes apply to all the tables describing attributes of keys:

The following table defines the attributes common to public key, private key and secret key classes, in addition to the common attributes defined for this object class:

Table 23, Common Key Attributes

Attribute	Data Type	Meaning
CKA_KEY_TYPE ^{1,5}	CK_KEY_TYPE	Type of key
CKA_ID ⁸	Byte array	Key identifier for key (default empty)
CKA_START_DATE ⁸	CK_DATE	Start date for the key (default empty)
CKA_END_DATE ⁸	CK_DATE	End date for the key (default empty)
CKA_DERIVE ⁸	CK_BBOOL	CK_TRUE if key supports key derivation (<i>i.e.</i> , if other keys can be derived from this one (default CK_FALSE))
CKA_LOCAL ^{2,4,6}	CK_BBOOL	CK_TRUE only if key was either <ul style="list-style-type: none"> generated locally (<i>i.e.</i>, on the token) with a C_GenerateKey or C_GenerateKeyPair call created with a C_CopyObject call as a copy of a key which had its CKA_LOCAL attribute set to CK_TRUE
CKA_KEY_GEN_MECHANISM ^{2,4,6}	CK_MECHANISM_TYPE	Identifier of the mechanism used to generate the key material.
CKA_ALLOWED_MECHANISMS	CK_MECHANISM_TYPE_PTR, pointer to a CK_MECHANISM_TYPE array	A list of mechanisms allowed to be used with this key. The number of mechanisms in the array is the <i>ulValueLen</i> component of the attribute divided by the size of CK_MECHANISM_TYPE.

Refer to Table 11 for footnotes

The **CKA_ID** field is intended to distinguish among multiple keys. In the case of public and private keys, this field assists in handling multiple keys held by the same subject; the key identifier for a public key and its corresponding private key should be the same. The key identifier should also be the same as for the corresponding certificate, if one exists. Cryptoki does not enforce these associations, however. (See Section 4.6 for further commentary.)

In the case of secret keys, the meaning of the **CKA_ID** attribute is up to the application.

Note that the **CKA_START_DATE** and **CKA_END_DATE** attributes are for reference only; Cryptoki does not attach any special meaning to them. In particular, it does not restrict usage of a key according to the dates; doing this is up to the application.

The **CKA_DERIVE** attribute has the value CK_TRUE if and only if it is possible to derive other keys from the key.

The **CKA_LOCAL** attribute has the value CK_TRUE if and only if the value of the key was originally generated on the token by a **C_GenerateKey** or **C_GenerateKeyPair** call.

The **CKA_KEY_GEN_MECHANISM** attribute identifies the key generation mechanism used to generate the key material. It contains a valid value only if the **CKA_LOCAL** attribute has the value CK_TRUE. If **CKA_LOCAL** has the value CK_FALSE, the value of the attribute is CK_UNAVAILABLE_INFORMATION.

4.8 Public key objects

Public key objects (object class **CKO_PUBLIC_KEY**) hold public keys. The following table defines the attributes common to all public keys, in addition to the common attributes defined for this object class:

Table 24, Common Public Key Attributes

Attribute	Data type	Meaning
CKA_SUBJECT ⁸	Byte array	DER-encoding of the key subject name (default empty)
CKA_ENCRYPT ⁸	CK_BBOOL	CK_TRUE if key supports encryption ⁹
CKA_VERIFY ⁸	CK_BBOOL	CK_TRUE if key supports verification where the signature is an appendix to the data ⁹
CKA_VERIFY_RECOVER ⁸	CK_BBOOL	CK_TRUE if key supports verification where the data is recovered from the signature ⁹
CKA_WRAP ⁸	CK_BBOOL	CK_TRUE if key supports wrapping (i.e., can be used to wrap other keys) ⁹
CKA_TRUSTED ¹⁰	CK_BBOOL	The key can be trusted for the application that it was created. The wrapping key can be used to wrap keys with CKA_WRAP_WITH_TRUSTED set to CK_TRUE.
CKA_WRAP_TEMPLATE	CK_ATTRIBUTE_PTR	For wrapping keys. The attribute template to match against any keys wrapped using this wrapping key. Keys that do not match cannot be wrapped. The number of attributes in the array is the <i>ulValueLen</i> component of the attribute divided by the size of CK_ATTRIBUTE.
CKA_PUBLIC_KEY_INFO	Byte array	DER-encoding of the SubjectPublicKeyInfo for this public key. (MAY be empty, DEFAULT derived from the underlying public key data)

1568 Refer to Table 11 for footnotes

1569 It is intended in the interests of interoperability that the subject name and key identifier for a public key will
1570 be the same as those for the corresponding certificate and private key. However, Cryptoki does not
1571 enforce this, and it is not required that the certificate and private key also be stored on the token.

1572 To map between ISO/IEC 9594-8 (X.509) **keyUsage** flags for public keys and the PKCS #11 attributes for
1573 public keys, use the following table.

1574 *Table 25, Mapping of X.509 key usage flags to Cryptoki attributes for public keys*

Key usage flags for public keys in X.509 public key certificates	Corresponding cryptoki attributes for public keys.
dataEncipherment	CKA_ENCRYPT
digitalSignature, keyCertSign, cRLSign	CKA_VERIFY
digitalSignature, keyCertSign, cRLSign	CKA_VERIFY_RECOVER
keyAgreement	CKA_DERIVE
keyEncipherment	CKA_WRAP
nonRepudiation	CKA_VERIFY
nonRepudiation	CKA_VERIFY_RECOVER

1575 The value of the CKA_PUBLIC_KEY_INFO attribute is the DER encoded value of SubjectPublicKeyInfo:

1576 SubjectPublicKeyInfo ::= SEQUENCE {
 1577 algorithm AlgorithmIdentifier,
 1578 subjectPublicKey BIT_STRING }

1579 The encodings for the subjectPublicKey field are specified in the description of the public key types in the
 1580 appropriate [PKCS11-Curr] document for the key types defined within this specification.

1581 4.9 Private key objects

1582 Private key objects (object class **CKO_PRIVATE_KEY**) hold private keys. The following table defines the
 1583 attributes common to all private keys, in addition to the common attributes defined for this object class:

1584 Table 26, Common Private Key Attributes

Attribute	Data type	Meaning
CKA_SUBJECT ⁸	Byte array	DER-encoding of certificate subject name (default empty)
CKA_SENSITIVE ^{8,11}	CK_BBOOL	CK_TRUE if key is sensitive ⁹
CKA_DECRYPT ⁸	CK_BBOOL	CK_TRUE if key supports decryption ⁹
CKA_SIGN ⁸	CK_BBOOL	CK_TRUE if key supports signatures where the signature is an appendix to the data ⁹
CKA_SIGN_RECOVER ⁸	CK_BBOOL	CK_TRUE if key supports signatures where the data can be recovered from the signature ⁹
CKA_UNWRAP ⁸	CK_BBOOL	CK_TRUE if key supports unwrapping (<i>i.e.</i> , can be used to unwrap other keys) ⁹
CKA_EXTRACTABLE ^{8,12}	CK_BBOOL	CK_TRUE if key is extractable and can be wrapped ⁹
CKA_ALWAYS_SENSITIVE ^{2,4,6}	CK_BBOOL	CK_TRUE if key has <i>always</i> had the CKA_SENSITIVE attribute set to CK_TRUE
CKA_NEVER_EXTRACTABLE ^{2,4,6}	CK_BBOOL	CK_TRUE if key has <i>never</i> had the CKA_EXTRACTABLE attribute set to CK_TRUE
CKA_WRAP_WITH_TRUSTED ¹¹	CK_BBOOL	CK_TRUE if the key can only be wrapped with a wrapping key that has CKA_TRUSTED set to CK_TRUE. Default is CK_FALSE.
CKA_UNWRAP_TEMPLATE	CK_ATTRIBUTE_PTR	For wrapping keys. The attribute template to apply to any keys unwrapped using this wrapping key. Any user supplied template is applied after this template as if the object has already been created. The number of attributes in the array is the <i>ulValueLen</i> component of the attribute divided by the size of CK_ATTRIBUTE.

Attribute	Data type	Meaning
CKA_ALWAYS_AUTHENTICATE	CK_BBOOL	If CK_TRUE, the user has to supply the PIN for each use (sign or decrypt) with the key. Default is CK_FALSE.
CKA_PUBLIC_KEY_INFO ⁸	Byte Array	DER-encoding of the SubjectPublicKeyInfo for the associated public key (MAY be empty; DEFAULT derived from the underlying private key data; MAY be manually set for specific key types; if set; MUST be consistent with the underlying private key data)

Refer to Table 11 for footnotes

It is intended in the interests of interoperability that the subject name and key identifier for a private key will be the same as those for the corresponding certificate and public key. However, this is not enforced by Cryptoki, and it is not required that the certificate and public key also be stored on the token.

If the **CKA_SENSITIVE** attribute is CK_TRUE, or if the **CKA_EXTRACTABLE** attribute is CK_FALSE, then certain attributes of the private key cannot be revealed in plaintext outside the token. Which attributes these are is specified for each type of private key in the attribute table in the section describing that type of key.

The **CKA_ALWAYS_AUTHENTICATE** attribute can be used to force re-authentication (i.e. force the user to provide a PIN) for each use of a private key. "Use" in this case means a cryptographic operation such as sign or decrypt. This attribute may only be set to CK_TRUE when **CKA_PRIVATE** is also CK_TRUE.

Re-authentication occurs by calling **C_Login** with *userType* set to **CKU_CONTEXT_SPECIFIC** immediately after a cryptographic operation using the key has been initiated (e.g. after **C_SignInit**). In this call, the actual user type is implicitly given by the usage requirements of the active key. If **C_Login** returns CKR_OK the user was successfully authenticated and this sets the active key in an authenticated state that lasts until the cryptographic operation has successfully or unsuccessfully been completed (e.g. by **C_Sign**, **C_SignFinal**,...). A return value CKR_PIN_INCORRECT from **C_Login** means that the user was denied permission to use the key and continuing the cryptographic operation will result in a behavior as if **C_Login** had not been called. In both of these cases the session state will remain the same, however repeated failed re-authentication attempts may cause the PIN to be locked. **C_Login** returns in this case CKR_PIN_LOCKED and this also logs the user out from the token. Failing or omitting to re-authenticate when CKA_ALWAYS_AUTHENTICATE is set to CK_TRUE will result in CKR_USER_NOT_LOGGED_IN to be returned from calls using the key. **C_Login** will return CKR_OPERATION_NOT_INITIALIZED, but the active cryptographic operation will not be affected, if an attempt is made to re-authenticate when CKA_ALWAYS_AUTHENTICATE is set to CK_FALSE.

The **CKA_PUBLIC_KEY_INFO** attribute represents the public key associated with this private key. The data it represents may either be stored as part of the private key data, or regenerated as needed from the private key.

If this attribute is supplied as part of a template for **C_CreateObject**, **C_CopyObject** or **C_SetAttributeValue** for a private key, the token MUST verify correspondence between the private key data and the public key data as supplied in **CKA_PUBLIC_KEY_INFO**. This can be done either by deriving a public key from the private key and comparing the values, or by doing a sign and verify operation. If there is a mismatch, the command SHALL return **CKR_ATTRIBUTE_VALUE_INVALID**. A token MAY choose not to support the **CKA_PUBLIC_KEY_INFO** attribute for commands which create new private keys. If it does not support the attribute, the command SHALL return **CKR_ATTRIBUTE_TYPE_INVALID**.

As a general guideline, private keys of any type SHOULD store sufficient information to retrieve the public key information. In particular, the RSA private key description has been modified in <this version> to add the CKA_PUBLIC_EXPONENT to the list of attributes required for an RSA private key. All other private

key types described in this specification contain sufficient information to recover the associated public key.

4.9.1 RSA private key objects

RSA private key objects (object class **CKO_PRIVATE_KEY**, key type **CKK_RSA**) hold RSA private keys. The following table defines the RSA private key object attributes, in addition to the common attributes defined for this object class:

Table 26, RSA Private Key Object Attributes

Attribute	Data type	Meaning
CKA_MODULUS ^{1,4,6}	Big integer	Modulus n
CKA_PUBLIC_EXPONENT ^{1,4,6}	Big integer	Public exponent e
CKA_PRIVATE_EXPONENT ^{1,4,6,7}	Big integer	Private exponent d
CKA_PRIME_1 ^{4,6,7}	Big integer	Prime p
CKA_PRIME_2 ^{4,6,7}	Big integer	Prime q
CKA_EXPONENT_1 ^{4,6,7}	Big integer	Private exponent d modulo $p-1$
CKA_EXPONENT_2 ^{4,6,7}	Big integer	Private exponent d modulo $q-1$
CKA_COEFFICIENT ^{4,6,7}	Big integer	CRT coefficient $q^{-1} \bmod p$

Refer to Table 10 for footnotes

Depending on the token, there may be limits on the length of the key components. See PKCS #1 for more information on RSA keys.

Tokens vary in what they actually store for RSA private keys. Some tokens store all of the above attributes, which can assist in performing rapid RSA computations. Other tokens might store only the **CKA_MODULUS** and **CKA_PRIVATE_EXPONENT** values. Effective with version 2.40, tokens **MUST** also store **CKA_PUBLIC_EXPONENT**. This permits the retrieval of sufficient data to reconstitute the associated public key.

Because of this, Cryptoki is flexible in dealing with RSA private key objects. When a token generates an RSA private key, it stores whichever of the fields in Table 26 it keeps track of. Later, if an application asks for the values of the key's various attributes, Cryptoki supplies values only for attributes whose values it can obtain (i.e., if Cryptoki is asked for the value of an attribute it cannot obtain, the request fails). Note that a Cryptoki implementation may or may not be able and/or willing to supply various attributes of RSA private keys which are not actually stored on the token. *E.g.*, if a particular token stores values only for the **CKA_PRIVATE_EXPONENT**, **CKA_PUBLIC_EXPONENT**, **CKA_PRIME_1**, and **CKA_PRIME_2** attributes, then Cryptoki is certainly *able* to report values for all the attributes above (since they can all be computed efficiently from these four values). However, a Cryptoki implementation may or may not actually do this extra computation. The only attributes from Table 26 for which a Cryptoki implementation is *required* to be able to return values are **CKA_MODULUS**, **CKA_PRIVATE_EXPONENT**, and **CKA_PUBLIC_EXPONENT**. A token **SHOULD** also be able to return **CKA_PUBLIC_KEY_INFO** for an RSA private key. See the general guidance for Private Keys above.

4.10 Secret key objects

Secret key objects (object class **CKO_SECRET_KEY**) hold secret keys. The following table defines the attributes common to all secret keys, in addition to the common attributes defined for this object class:

Table 27, Common Secret Key Attributes

Attribute	Data type	Meaning
CKA_SENSITIVE ^{8,11}	CK_BBOOL	CK_TRUE if object is sensitive (default CK_FALSE)
CKA_ENCRYPT ⁸	CK_BBOOL	CK_TRUE if key supports encryption ⁹
CKA_DECRYPT ⁸	CK_BBOOL	CK_TRUE if key supports decryption ⁹
CKA_SIGN ⁸	CK_BBOOL	CK_TRUE if key supports signatures (<i>i.e.</i> , authentication codes) where the signature is an appendix to the data ⁹
CKA_VERIFY ⁸	CK_BBOOL	CK_TRUE if key supports verification (<i>i.e.</i> , of authentication codes) where the signature is an appendix to the data ⁹
CKA_WRAP ⁸	CK_BBOOL	CK_TRUE if key supports wrapping (<i>i.e.</i> , can be used to wrap other keys) ⁹
CKA_UNWRAP ⁸	CK_BBOOL	CK_TRUE if key supports unwrapping (<i>i.e.</i> , can be used to unwrap other keys) ⁹
CKA_EXTRACTABLE ^{8,12}	CK_BBOOL	CK_TRUE if key is extractable and can be wrapped ⁹
CKA_ALWAYS_SENSITIVE ^{2,4,6}	CK_BBOOL	CK_TRUE if key has <i>always</i> had the CKA_SENSITIVE attribute set to CK_TRUE
CKA_NEVER_EXTRACTABLE ^{2,4,6}	CK_BBOOL	CK_TRUE if key has <i>never</i> had the CKA_EXTRACTABLE attribute set to CK_TRUE
CKA_CHECK_VALUE	Byte array	Key checksum
CKA_WRAP_WITH_TRUSTED ¹¹	CK_BBOOL	CK_TRUE if the key can only be wrapped with a wrapping key that has CKA_TRUSTED set to CK_TRUE. Default is CK_FALSE.
CKA_TRUSTED ¹⁰	CK_BBOOL	The wrapping key can be used to wrap keys with CKA_WRAP_WITH_TRUSTED set to CK_TRUE.
CKA_WRAP_TEMPLATE	CK_ATTRIBUTE_PTR	For wrapping keys. The attribute template to match against any keys wrapped using this wrapping key. Keys that do not match cannot be wrapped. The number of attributes in the array is the <i>ulValueLen</i> component of the attribute divided by the size of CK_ATTRIBUTE

Attribute	Data type	Meaning
CKA_UNWRAP_TEMPLATE	CK_ATTRIBUTE_PTR	For wrapping keys. The attribute template to apply to any keys unwrapped using this wrapping key. Any user supplied template is applied after this template as if the object has already been created. The number of attributes in the array is the <i>ulValueLen</i> component of the attribute divided by the size of CK_ATTRIBUTE.

Refer to Table 11 for footnotes

If the **CKA_SENSITIVE** attribute is CK_TRUE, or if the **CKA_EXTRACTABLE** attribute is CK_FALSE, then certain attributes of the secret key cannot be revealed in plaintext outside the token. Which attributes these are is specified for each type of secret key in the attribute table in the section describing that type of key.

The key check value (KCV) attribute for symmetric key objects to be called **CKA_CHECK_VALUE**, of type byte array, length 3 bytes, operates like a fingerprint, or checksum of the key. They are intended to be used to cross-check symmetric keys against other systems where the same key is shared, and as a validity check after manual key entry or restore from backup. Refer to object definitions of specific key types for KCV algorithms.

Properties:

1. For two keys that are cryptographically identical the value of this attribute should be identical.
2. CKA_CHECK_VALUE should not be usable to obtain any part of the key value.
3. Non-uniqueness. Two different keys can have the same CKA_CHECK_VALUE. This is unlikely (the probability can easily be calculated) but possible.

The attribute is optional, but if supported, regardless of how the key object is created or derived, the value of the attribute is always supplied. It SHALL be supplied even if the encryption operation for the key is forbidden (i.e. when CKA_ENCRYPT is set to CK_FALSE).

If a value is supplied in the application template (allowed but never necessary) then, if supported, it MUST match what the library calculates it to be or the library returns a CKR_ATTRIBUTE_VALUE_INVALID. If the library does not support the attribute then it should ignore it. Allowing the attribute in the template this way does no harm and allows the attribute to be treated like any other attribute for the purposes of key wrap and unwrap where the attributes are preserved also.

The generation of the KCV may be prevented by the application supplying the attribute in the template as a no-value (0 length) entry. The application can query the value at any time like any other attribute using C_GetAttributeValue. C_SetAttributeValue may be used to destroy the attribute, by supplying no-value.

Unless otherwise specified for the object definition, the value of this attribute is derived from the key object by taking the first three bytes of an encryption of a single block of null (0x00) bytes, using the default cipher and mode (e.g. ECB) associated with the key type of the secret key object.

4.11 Domain parameter objects

4.11.1 Definitions

This section defines the object class CKO_DOMAIN_PARAMETERS for type CK_OBJECT_CLASS as used in the CKA_CLASS attribute of objects.

4.11.2 Overview

This object class was created to support the storage of certain algorithm's extended parameters. DSA and DH both use domain parameters in the key-pair generation step. In particular, some libraries support the generation of domain parameters (originally out of scope for PKCS11) so the object class was added.

To use a domain parameter object you MUST extract the attributes into a template and supply them (still in the template) to the corresponding key-pair generation function.

Domain parameter objects (object class **CKO_DOMAIN_PARAMETERS**) hold public domain parameters.

The following table defines the attributes common to domain parameter objects in addition to the common attributes defined for this object class:

Table 28, Common Domain Parameter Attributes

Attribute	Data Type	Meaning
CKA_KEY_TYPE ¹	CK_KEY_TYPE	Type of key the domain parameters can be used to generate.
CKA_LOCAL ^{2,4}	CK_BBOOL	CK_TRUE only if domain parameters were either <ul style="list-style-type: none">generated locally (<i>i.e.</i>, on the token) with a C_GenerateKeycreated with a C_CopyObject call as a copy of domain parameters which had its CKA_LOCAL attribute set to CK_TRUE

¹ Refer to Table 11 for footnotes

The **CKA_LOCAL** attribute has the value CK_TRUE if and only if the values of the domain parameters were originally generated on the token by a **C_GenerateKey** call.

4.12 Mechanism objects

4.12.1 Definitions

This section defines the object class CKO_MECHANISM for type CK_OBJECT_CLASS as used in the CKA_CLASS attribute of objects.

4.12.2 Overview

Mechanism objects provide information about mechanisms supported by a device beyond that given by the **CK_MECHANISM_INFO** structure.

When searching for objects using **C_FindObjectsInit** and **C_FindObjects**, mechanism objects are not returned unless the **CKA_CLASS** attribute in the template has the value **CKO_MECHANISM**. This protects applications written to previous versions of Cryptoki from finding objects that they do not understand.

Table 29, Common Mechanism Attributes

Attribute	Data Type	Meaning
CKA_MECHANISM_TYPE	CK_MECHANISM_TYPE	The type of mechanism object

The **CKA_MECHANISM_TYPE** attribute may not be set.

4.13 Profile objects

4.13.1 Definitions

This section defines the object class CKO_PROFILE for type CK_OBJECT_CLASS as used in the CKA_CLASS attribute of objects.

4.13.2 Overview

Profile objects (object class CKO_PROFILE) describe which PKCS #11 profiles the token implements. Profiles are defined in the OASIS PKCS #11 Cryptographic Token Interface Profiles document. A given token can contain more than one profile ID.. The following table lists the attributes supported by profile objects, in addition to the common attributes defined for this object class:

Table 27, Profile Object Attributes

Attribute	Data type	Meaning
CKA_PROFILE_ID	CK_PROFILE_ID	ID of the supported profile.

The **CKA_PROFILE** attribute identifies a profile that the token supports.

5 Functions

Cryptoki's functions are organized into the following categories:

- general-purpose functions (4 functions)
- slot and token management functions (9 functions)
- session management functions (8 functions)
- object management functions (9 functions)
- encryption functions (4 functions)
- message-based encryption functions (5 functions)
- decryption functions (4 functions)
- message digesting functions (5 functions)
- signing and MACing functions (6 functions)
- functions for verifying signatures and MACs (6 functions)
- dual-purpose cryptographic functions (4 functions)
- key management functions (5 functions)
- random number generation functions (2 functions)
- parallel function management functions (2 functions)

In addition to these functions, Cryptoki can use application-supplied callback functions to notify an application of certain events, and can also use application-supplied functions to handle mutex objects for safe multi-threaded library access.

The Cryptoki API functions are presented in the following table:

Table 30, Summary of Cryptoki Functions

Category	Function	Description
General purpose functions	C_Initialize	initializes Cryptoki
	C_Finalize	clean up miscellaneous Cryptoki-associated resources
	C_GetInfo	obtains general information about Cryptoki
	C_GetFunctionList	obtains entry points of Cryptoki library functions
	C_GetInterfaceList	obtains list of interfaces supported by Cryptoki library
	C_GetInterface	obtains interface specific entry points to Cryptoki library functions
Slot and token management functions	C_GetSlotList	obtains a list of slots in the system
	C_GetSlotInfo	obtains information about a particular slot
	C_GetTokenInfo	obtains information about a particular token
	C_WaitForSlotEvent	waits for a slot event (token insertion, removal, etc.) to occur
	C_GetMechanismList	obtains a list of mechanisms supported by a token
	C_GetMechanismInfo	obtains information about a particular mechanism
	C_InitToken	initializes a token
	C_InitPIN	initializes the normal user's PIN

Category	Function	Description
	C_SetPIN	modifies the PIN of the current user
Session management functions	C_OpenSession	opens a connection between an application and a particular token or sets up an application callback for token insertion
	C_CloseSession	closes a session
	C_CloseAllSessions	closes all sessions with a token
	C_GetSessionInfo	obtains information about the session
	C_SessionCancel	terminates active session based operations
	C_GetOperationState	obtains the cryptographic operations state of a session
	C_SetOperationState	sets the cryptographic operations state of a session
	C_Login	logs into a token
	C_LoginUser	??????
	C_Logout	logs out from a token
Object management functions	C_CreateObject	creates an object
	C_CopyObject	creates a copy of an object
	C_DestroyObject	destroys an object
	C_GetObjectSize	obtains the size of an object in bytes
	C_GetAttributeValue	obtains an attribute value of an object
	C_SetAttributeValue	modifies an attribute value of an object
	C_FindObjectsInit	initializes an object search operation
	C_FindObjects	continues an object search operation
	C_FindObjectsFinal	finishes an object search operation
Encryption functions	C_EncryptInit	initializes an encryption operation
	C_Encrypt	encrypts single-part data
	C_EncryptUpdate	continues a multiple-part encryption operation
	C_EncryptFinal	finishes a multiple-part encryption operation
Message-based Encryption Functions	C_MessageEncryptInit	initializes a message-based encryption process
	C_EncryptMessage	encrypts a single-part message
	C_EncryptMessageBegin	begins a multiple-part message encryption operation
	C_EncryptMessageNext	continues or finishes a multiple-part message encryption operation
	C_MessageEncryptFinal	finishes a message-based encryption process
Decryption Functions	C_DecryptInit	initializes a decryption operation
	C_Decrypt	decrypts single-part encrypted data
	C_DecryptUpdate	continues a multiple-part decryption operation
	C_DecryptFinal	finishes a multiple-part decryption operation
Message-based Decryption Functions	C_MessageDecryptInit	initializes a message decryption operation
	C_DecryptMessage	decrypts single-part data
	C_DecryptMessageBegin	starts a multiple-part message decryption operation

Category	Function	Description
	C_DecryptMessageNext	Continues and finishes a multiple-part message decryption operation
	C_MessageDecryptFinal	finishes a message decryption operation
Message Digesting Functions	C_DigestInit	initializes a message-digesting operation
	C_Digest	digests single-part data
	C_DigestUpdate	continues a multiple-part digesting operation
	C_DigestKey	digests a key
	C_DigestFinal	finishes a multiple-part digesting operation
Signing and MACing functions	C_SignInit	initializes a signature operation
	C_Sign	signs single-part data
	C_SignUpdate	continues a multiple-part signature operation
	C_SignFinal	finishes a multiple-part signature operation
	C_SignRecoverInit	initializes a signature operation, where the data can be recovered from the signature
	C_SignRecover	signs single-part data, where the data can be recovered from the signature
Message-based Signature functions	C_MessageSignInit	initializes a message signature operation
	C_SignMessage	signs single-part data
	C_SignMessageBegin	starts a multiple-part message signature operation
	C_SignMessageNext	continues and finishes a multiple-part message signature operation
	C_MessageSignFinal	finishes a message signature operation
Functions for verifying signatures and MACs	C_VerifyInit	initializes a verification operation
	C_Verify	verifies a signature on single-part data
	C_VerifyUpdate	continues a multiple-part verification operation
	C_VerifyFinal	finishes a multiple-part verification operation
	C_VerifyRecoverInit	initializes a verification operation where the data is recovered from the signature
	C_VerifyRecover	verifies a signature on single-part data, where the data is recovered from the signature
Message-based Functions for verifying signatures and MACs	C_MessageVerifyInit	initializes a message verification operation
	C_VerifyMessage	verifies single-part data
	C_VerifyMessageBegin	starts a multiple-part message verification operation
	C_VerifyMessageNext	continues and finishes a multiple-part message verification operation
	C_MessageVerifyFinal	finishes a message verification operation
Dual-purpose cryptographic functions	C_DigestEncryptUpdate	continues simultaneous multiple-part digesting and encryption operations
	C_DecryptDigestUpdate	continues simultaneous multiple-part decryption and digesting operations
	C_SignEncryptUpdate	continues simultaneous multiple-part signature and encryption operations
	C_DecryptVerifyUpdate	continues simultaneous multiple-part decryption and verification operations

Category	Function	Description
Key management functions	C_GenerateKey	generates a secret key
	C_GenerateKeyPair	generates a public-key/private-key pair
	C_WrapKey	wraps (encrypts) a key
	C_UnwrapKey	unwraps (decrypts) a key
	C_DeriveKey	derives a key from a base key
Random number generation functions	C_SeedRandom	mixes in additional seed material to the random number generator
	C_GenerateRandom	generates random data
Parallel function management functions	C_GetFunctionStatus	legacy function which always returns CKR_FUNCTION_NOT_PARALLEL
	C_CancelFunction	legacy function which always returns CKR_FUNCTION_NOT_PARALLEL
Callback function		application-supplied function to process notifications from Cryptoki

Execution of a Cryptoki function call is in general an all-or-nothing affair, *i.e.*, a function call accomplishes either its entire goal, or nothing at all.

- If a Cryptoki function executes successfully, it returns the value CKR_OK.
- If a Cryptoki function does not execute successfully, it returns some value other than CKR_OK, and the token is in the same state as it was in prior to the function call. If the function call was supposed to modify the contents of certain memory addresses on the host computer, these memory addresses may have been modified, despite the failure of the function.
- In unusual (and extremely unpleasant!) circumstances, a function can fail with the return value CKR_GENERAL_ERROR. When this happens, the token and/or host computer may be in an inconsistent state, and the goals of the function may have been partially achieved.

There are a small number of Cryptoki functions whose return values do not behave precisely as described above; these exceptions are documented individually with the description of the functions themselves.

A Cryptoki library need not support every function in the Cryptoki API. However, even an unsupported function **MUST** have a “stub” in the library which simply returns the value CKR_FUNCTION_NOT_SUPPORTED. The function’s entry in the library’s **CK_FUNCTION_LIST** structure (as obtained by **C_GetFunctionList**) should point to this stub function (see Section 3.6).

5.1 Function return values

The Cryptoki interface possesses a large number of functions and return values. In Section 5.1, we enumerate the various possible return values for Cryptoki functions; most of the remainder of Section 5.1 details the behavior of Cryptoki functions, including what values each of them may return.

Because of the complexity of the Cryptoki specification, it is recommended that Cryptoki applications attempt to give some leeway when interpreting Cryptoki functions’ return values. We have attempted to specify the behavior of Cryptoki functions as completely as was feasible; nevertheless, there are presumably some gaps. For example, it is possible that a particular error code which might apply to a particular Cryptoki function is unfortunately not actually listed in the description of that function as a possible error code. It is conceivable that the developer of a Cryptoki library might nevertheless permit his/her implementation of that function to return that error code. It would clearly be somewhat ungraceful if a Cryptoki application using that library were to terminate by abruptly dumping core upon receiving that error code for that function. It would be far preferable for the application to examine the function’s return value, see that it indicates some sort of error (even if the application doesn’t know precisely *what* kind of error), and behave accordingly.

See Section 5.1.8 for some specific details on how a developer might attempt to make an application that accommodates a range of behaviors from Cryptoki libraries.

5.1.1 Universal Cryptoki function return values

Any Cryptoki function can return any of the following values:

- **CKR_GENERAL_ERROR**: Some horrible, unrecoverable error has occurred. In the worst case, it is possible that the function only partially succeeded, and that the computer and/or token is in an inconsistent state.
- **CKR_HOST_MEMORY**: The computer that the Cryptoki library is running on has insufficient memory to perform the requested function.
- **CKR_FUNCTION_FAILED**: The requested function could not be performed, but detailed information about why not is not available in this error return. If the failed function uses a session, it is possible that the **CK_SESSION_INFO** structure that can be obtained by calling **C_GetSessionInfo** will hold useful information about what happened in its *ulDeviceError* field. In any event, although the function call failed, the situation is not necessarily totally hopeless, as it is likely to be when **CKR_GENERAL_ERROR** is returned. Depending on what the root cause of the error actually was, it is possible that an attempt to make the exact same function call again would succeed.
- **CKR_OK**: The function executed successfully. Technically, **CKR_OK** is not *quite* a “universal” return value; in particular, the legacy functions **C_GetFunctionStatus** and **C_CancelFunction** (see Section 5.20) cannot return **CKR_OK**.

The relative priorities of these errors are in the order listed above, *e.g.*, if either of **CKR_GENERAL_ERROR** or **CKR_HOST_MEMORY** would be an appropriate error return, then **CKR_GENERAL_ERROR** should be returned.

5.1.2 Cryptoki function return values for functions that use a session handle

Any Cryptoki function that takes a session handle as one of its arguments (i.e., any Cryptoki function except for **C_Initialize**, **C_Finalize**, **C_GetInfo**, **C_GetFunctionList**, **C_GetSlotList**, **C_GetSlotInfo**, **C_GetTokenInfo**, **C_WaitForSlotEvent**, **C_GetMechanismList**, **C_GetMechanismInfo**, **C_InitToken**, **C_OpenSession**, and **C_CloseAllSessions**) can return the following values:

- **CKR_SESSION_HANDLE_INVALID**: The specified session handle was invalid *at the time that the function was invoked*. Note that this can happen if the session's token is removed before the function invocation, since removing a token closes all sessions with it.
- **CKR_DEVICE_REMOVED**: The token was removed from its slot *during the execution of the function*.
- **CKR_SESSION_CLOSED**: The session was closed *during the execution of the function*. Note that, as stated in **[PKCS11-UG]**, the behavior of Cryptoki is *undefined* if multiple threads of an application attempt to access a common Cryptoki session simultaneously. Therefore, there is actually no guarantee that a function invocation could ever return the value **CKR_SESSION_CLOSED**. An example of multiple threads accessing a common session simultaneously is where one thread is using a session when another thread closes that same session.

The relative priorities of these errors are in the order listed above, *e.g.*, if either of **CKR_SESSION_HANDLE_INVALID** or **CKR_DEVICE_REMOVED** would be an appropriate error return, then **CKR_SESSION_HANDLE_INVALID** should be returned.

In practice, it is often not crucial (or possible) for a Cryptoki library to be able to make a distinction between a token being removed *before* a function invocation and a token being removed *during* a function execution.

5.1.3 Cryptoki function return values for functions that use a token

Any Cryptoki function that uses a particular token (*i.e.*, any Cryptoki function except for **C_Initialize**, **C_Finalize**, **C_GetInfo**, **C_GetFunctionList**, **C_GetSlotList**, **C_GetSlotInfo**, or **C_WaitForSlotEvent**) can return any of the following values:

- **CKR_DEVICE_MEMORY**: The token does not have sufficient memory to perform the requested function.
- **CKR_DEVICE_ERROR**: Some problem has occurred with the token and/or slot. This error code can be returned by more than just the functions mentioned above; in particular, it is possible for **C_GetSlotInfo** to return **CKR_DEVICE_ERROR**.
- **CKR_TOKEN_NOT_PRESENT**: The token was not present in its slot *at the time that the function was invoked*.
- **CKR_DEVICE_REMOVED**: The token was removed from its slot *during the execution of the function*.

The relative priorities of these errors are in the order listed above, *e.g.*, if either of **CKR_DEVICE_MEMORY** or **CKR_DEVICE_ERROR** would be an appropriate error return, then **CKR_DEVICE_MEMORY** should be returned.

In practice, it is often not critical (or possible) for a Cryptoki library to be able to make a distinction between a token being removed *before* a function invocation and a token being removed *during* a function execution.

5.1.4 Special return value for application-supplied callbacks

There is a special-purpose return value which is not returned by any function in the actual Cryptoki API, but which may be returned by an application-supplied callback function. It is:

- **CKR_CANCEL**: When a function executing in serial with an application decides to give the application a chance to do some work, it calls an application-supplied function with a **CKN_SURRENDER** callback (see Section 5.21). If the callback returns the value **CKR_CANCEL**, then the function aborts and returns **CKR_FUNCTION_CANCELED**.

5.1.5 Special return values for mutex-handling functions

There are two other special-purpose return values which are not returned by any actual Cryptoki functions. These values may be returned by application-supplied mutex-handling functions, and they may safely be ignored by application developers who are not using their own threading model. They are:

- **CKR_MUTEX_BAD**: This error code can be returned by mutex-handling functions that are passed a bad mutex object as an argument. Unfortunately, it is possible for such a function not to recognize a bad mutex object. There is therefore no guarantee that such a function will successfully detect bad mutex objects and return this value.
- **CKR_MUTEX_NOT_LOCKED**: This error code can be returned by mutex-unlocking functions. It indicates that the mutex supplied to the mutex-unlocking function was not locked.

5.1.6 All other Cryptoki function return values

Descriptions of the other Cryptoki function return values follow. Except as mentioned in the descriptions of particular error codes, there are in general no particular priorities among the errors listed below, *i.e.*, if more than one error code might apply to an execution of a function, then the function may return any applicable error code.

- **CKR_ACTION_PROHIBITED**: This value can only be returned by **C_CopyObject**, **C_SetAttributeValue** and **C_DestroyObject**. It denotes that the action may not be taken, either because of underlying policy restrictions on the token, or because the object has the relevant **CKA_COPYABLE**, **CKA_MODIFIABLE** or **CKA_DESTROYABLE** policy attribute set to **CK_FALSE**.
- **CKR_ARGUMENTS_BAD**: This is a rather generic error code which indicates that the arguments supplied to the Cryptoki function were in some way not appropriate.

- 1872 • CKR_ATTRIBUTE_READ_ONLY: An attempt was made to set a value for an attribute which may not
1873 be set by the application, or which may not be modified by the application. See Section 4.1 for more
1874 information.
- 1875 • CKR_ATTRIBUTE_SENSITIVE: An attempt was made to obtain the value of an attribute of an object
1876 which cannot be satisfied because the object is either sensitive or un-extractable.
- 1877 • CKR_ATTRIBUTE_TYPE_INVALID: An invalid attribute type was specified in a template. See
1878 Section 4.1 for more information.
- 1879 • CKR_ATTRIBUTE_VALUE_INVALID: An invalid value was specified for a particular attribute in a
1880 template. See Section 4.1 for more information.
- 1881 • CKR_BUFFER_TOO_SMALL: The output of the function is too large to fit in the supplied buffer.
- 1882 • CKR_CANT_LOCK: This value can only be returned by **C_Initialize**. It means that the type of locking
1883 requested by the application for thread-safety is not available in this library, and so the application
1884 cannot make use of this library in the specified fashion.
- 1885 • CKR_CRYPTOKI_ALREADY_INITIALIZED: This value can only be returned by **C_Initialize**. It
1886 means that the Cryptoki library has already been initialized (by a previous call to **C_Initialize** which
1887 did not have a matching **C_Finalize** call).
- 1888 • CKR_CRYPTOKI_NOT_INITIALIZED: This value can be returned by any function other than
1889 **C_Initialize**, **C_GetFunctionList**, **C_GetInterfaceList** and **C_GetInterface**. It indicates that the
1890 function cannot be executed because the Cryptoki library has not yet been initialized by a call to
1891 **C_Initialize**.
- 1892 • CKR_CURVE_NOT_SUPPORTED: This curve is not supported by this token. Used with Elliptic
1893 Curve mechanisms.
- 1894 • CKR_DATA_INVALID: The plaintext input data to a cryptographic operation is invalid. This return
1895 value has lower priority than CKR_DATA_LEN_RANGE.
- 1896 • CKR_DATA_LEN_RANGE: The plaintext input data to a cryptographic operation has a bad length.
1897 Depending on the operation's mechanism, this could mean that the plaintext data is too short, too
1898 long, or is not a multiple of some particular block size. This return value has higher priority than
1899 CKR_DATA_INVALID.
- 1900 • CKR_DOMAIN_PARAMS_INVALID: Invalid or unsupported domain parameters were supplied to the
1901 function. Which representation methods of domain parameters are supported by a given mechanism
1902 can vary from token to token.
- 1903 • CKR_ENCRYPTED_DATA_INVALID: The encrypted input to a decryption operation has been
1904 determined to be invalid ciphertext. This return value has lower priority than
1905 CKR_ENCRYPTED_DATA_LEN_RANGE.
- 1906 • CKR_ENCRYPTED_DATA_LEN_RANGE: The ciphertext input to a decryption operation has been
1907 determined to be invalid ciphertext solely on the basis of its length. Depending on the operation's
1908 mechanism, this could mean that the ciphertext is too short, too long, or is not a multiple of some
1909 particular block size. This return value has higher priority than CKR_ENCRYPTED_DATA_INVALID.
- 1910 • CKR_EXCEEDED_MAX_ITERATIONS: An iterative algorithm (for key pair generation, domain
1911 parameter generation etc.) failed because we have exceeded the maximum number of iterations.
1912 This error code has precedence over CKR_FUNCTION_FAILED. Examples of iterative algorithms
1913 include DSA signature generation (retry if either $r = 0$ or $s = 0$) and generation of DSA primes p and q
1914 specified in FIPS 186-4.
- 1915 • CKR_FIPS_SELF_TEST_FAILED: A FIPS 140-2 power-up self-test or conditional self-test failed.
1916 The token entered an error state. Future calls to cryptographic functions on the token will return
1917 CKR_GENERAL_ERROR. CKR_FIPS_SELF_TEST_FAILED has a higher precedence over
1918 CKR_GENERAL_ERROR. This error may be returned by **C_Initialize**, if a power-up self-test failed,
1919 by **C_GenerateRandom** or **C_SeedRandom**, if the continuous random number generator test failed,
1920 or by **C_GenerateKeyPair**, if the pair-wise consistency test failed.

- 1921 • CKR_FUNCTION_CANCELED: The function was canceled in mid-execution. This happens to a
1922 cryptographic function if the function makes a **CKN_SURRENDER** application callback which returns
1923 CKR_CANCEL (see CKR_CANCEL). It also happens to a function that performs PIN entry through a
1924 protected path. The method used to cancel a protected path PIN entry operation is device dependent.
- 1925 • CKR_FUNCTION_NOT_PARALLEL: There is currently no function executing in parallel in the
1926 specified session. This is a legacy error code which is only returned by the legacy functions
1927 **C_GetFunctionStatus** and **C_CancelFunction**.
- 1928 • CKR_FUNCTION_NOT_SUPPORTED: The requested function is not supported by this Cryptoki
1929 library. Even unsupported functions in the Cryptoki API should have a "stub" in the library; this stub
1930 should simply return the value CKR_FUNCTION_NOT_SUPPORTED.
- 1931 • CKR_FUNCTION_REJECTED: The signature request is rejected by the user.
- 1932 • CKR_INFORMATION_SENSITIVE: The information requested could not be obtained because the
1933 token considers it sensitive, and is not able or willing to reveal it.
- 1934 • CKR_KEY_CHANGED: This value is only returned by **C_SetOperationState**. It indicates that one of
1935 the keys specified is not the same key that was being used in the original saved session.
- 1936 • CKR_KEY_FUNCTION_NOT_PERMITTED: An attempt has been made to use a key for a
1937 cryptographic purpose that the key's attributes are not set to allow it to do. For example, to use a key
1938 for performing encryption, that key MUST have its **CKA_ENCRYPT** attribute set to CK_TRUE (the
1939 fact that the key MUST have a **CKA_ENCRYPT** attribute implies that the key cannot be a private
1940 key). This return value has lower priority than CKR_KEY_TYPE_INCONSISTENT.
- 1941 • CKR_KEY_HANDLE_INVALID: The specified key handle is not valid. It may be the case that the
1942 specified handle is a valid handle for an object which is not a key. We reiterate here that 0 is never a
1943 valid key handle.
- 1944 • CKR_KEY_INDIGESTIBLE: This error code can only be returned by **C_DigestKey**. It indicates that
1945 the value of the specified key cannot be digested for some reason (perhaps the key isn't a secret key,
1946 or perhaps the token simply can't digest this kind of key).
- 1947 • CKR_KEY_NEEDED: This value is only returned by **C_SetOperationState**. It indicates that the
1948 session state cannot be restored because **C_SetOperationState** needs to be supplied with one or
1949 more keys that were being used in the original saved session.
- 1950 • CKR_KEY_NOT_NEEDED: An extraneous key was supplied to **C_SetOperationState**. For
1951 example, an attempt was made to restore a session that had been performing a message digesting
1952 operation, and an encryption key was supplied.
- 1953 • CKR_KEY_NOT_WRAPPABLE: Although the specified private or secret key does not have its
1954 CKA_EXTRACTABLE attribute set to CK_FALSE, Cryptoki (or the token) is unable to wrap the key as
1955 requested (possibly the token can only wrap a given key with certain types of keys, and the wrapping
1956 key specified is not one of these types). Compare with CKR_KEY_UNEXTRACTABLE.
- 1957 • CKR_KEY_SIZE_RANGE: Although the requested keyed cryptographic operation could in principle
1958 be carried out, this Cryptoki library (or the token) is unable to actually do it because the supplied key's
1959 size is outside the range of key sizes that it can handle.
- 1960 • CKR_KEY_TYPE_INCONSISTENT: The specified key is not the correct type of key to use with the
1961 specified mechanism. This return value has a higher priority than
1962 CKR_KEY_FUNCTION_NOT_PERMITTED.
- 1963 • CKR_KEY_UNEXTRACTABLE: The specified private or secret key can't be wrapped because its
1964 CKA_EXTRACTABLE attribute is set to CK_FALSE. Compare with CKR_KEY_NOT_WRAPPABLE.
- 1965 • CKR_LIBRARY_LOAD_FAILED: The Cryptoki library could not load a dependent shared library.
- 1966 • CKR_MECHANISM_INVALID: An invalid mechanism was specified to the cryptographic operation.
1967 This error code is an appropriate return value if an unknown mechanism was specified or if the
1968 mechanism specified cannot be used in the selected token with the selected function.

- 1969 • CKR_MECHANISM_PARAM_INVALID: Invalid parameters were supplied to the mechanism specified
1970 to the cryptographic operation. Which parameter values are supported by a given mechanism can
1971 vary from token to token.
- 1972 • CKR_NEED_TO_CREATE_THREADS: This value can only be returned by **C_Initialize**. It is
1973 returned when two conditions hold:
 - 1974 1. The application called **C_Initialize** in a way which tells the Cryptoki library that application
1975 threads executing calls to the library cannot use native operating system methods to spawn new
1976 threads.
 - 1977 2. The library cannot function properly without being able to spawn new threads in the above
1978 fashion.
- 1979 • CKR_NO_EVENT: This value can only be returned by **C_GetSlotEvent**. It is returned when
1980 **C_GetSlotEvent** is called in non-blocking mode and there are no new slot events to return.
- 1981 • CKR_OBJECT_HANDLE_INVALID: The specified object handle is not valid. We reiterate here that 0
1982 is never a valid object handle.
- 1983 • CKR_OPERATION_ACTIVE: There is already an active operation (or combination of active
1984 operations) which prevents Cryptoki from activating the specified operation. For example, an active
1985 object-searching operation would prevent Cryptoki from activating an encryption operation with
1986 **C_EncryptInit**. Or, an active digesting operation and an active encryption operation would prevent
1987 Cryptoki from activating a signature operation. Or, on a token which doesn't support simultaneous
1988 dual cryptographic operations in a session (see the description of the
1989 **CKF_DUAL_CRYPTO_OPERATIONS** flag in the **CK_TOKEN_INFO** structure), an active signature
1990 operation would prevent Cryptoki from activating an encryption operation.
- 1991 • CKR_OPERATION_NOT_INITIALIZED: There is no active operation of an appropriate type in the
1992 specified session. For example, an application cannot call **C_Encrypt** in a session without having
1993 called **C_EncryptInit** first to activate an encryption operation.
- 1994 • CKR_PIN_EXPIRED: The specified PIN has expired, and the requested operation cannot be carried
1995 out unless **C_SetPIN** is called to change the PIN value. Whether or not the normal user's PIN on a
1996 token ever expires varies from token to token.
- 1997 • CKR_PIN_INCORRECT: The specified PIN is incorrect, *i.e.*, does not match the PIN stored on the
1998 token. More generally-- when authentication to the token involves something other than a PIN-- the
1999 attempt to authenticate the user has failed.
- 2000 • CKR_PIN_INVALID: The specified PIN has invalid characters in it. This return code only applies to
2001 functions which attempt to set a PIN.
- 2002 • CKR_PIN_LEN_RANGE: The specified PIN is too long or too short. This return code only applies to
2003 functions which attempt to set a PIN.
- 2004 • CKR_PIN_LOCKED: The specified PIN is "locked", and cannot be used. That is, because some
2005 particular number of failed authentication attempts has been reached, the token is unwilling to permit
2006 further attempts at authentication. Depending on the token, the specified PIN may or may not remain
2007 locked indefinitely.
- 2008 • CKR_PIN_TOO_WEAK: The specified PIN is too weak so that it could be easy to guess. If the PIN is
2009 too short, **CKR_PIN_LEN_RANGE** should be returned instead. This return code only applies to
2010 functions which attempt to set a PIN.
- 2011 • CKR_PUBLIC_KEY_INVALID: The public key fails a public key validation. For example, an EC
2012 public key fails the public key validation specified in Section 5.2.2 of ANSI X9.62. This error code may
2013 be returned by **C_CreateObject**, when the public key is created, or by **C_VerifyInit** or
2014 **C_VerifyRecoverInit**, when the public key is used. It may also be returned by **C_DeriveKey**, in
2015 preference to **CKR_MECHANISM_PARAM_INVALID**, if the other party's public key specified in the
2016 mechanism's parameters is invalid.
- 2017 • CKR_RANDOM_NO_RNG: This value can be returned by **C_SeedRandom** and
2018 **C_GenerateRandom**. It indicates that the specified token doesn't have a random number generator.
2019 This return value has higher priority than **CKR_RANDOM_SEED_NOT_SUPPORTED**.

- 2020 • CKR_RANDOM_SEED_NOT_SUPPORTED: This value can only be returned by **C_SeedRandom**.
 2021 It indicates that the token's random number generator does not accept seeding from an application.
 2022 This return value has lower priority than CKR_RANDOM_NO_RNG.
- 2023 • CKR_SAVED_STATE_INVALID: This value can only be returned by **C_SetOperationState**. It
 2024 indicates that the supplied saved cryptographic operations state is invalid, and so it cannot be
 2025 restored to the specified session.
- 2026 • CKR_SESSION_COUNT: This value can only be returned by **C_OpenSession**. It indicates that the
 2027 attempt to open a session failed, either because the token has too many sessions already open, or
 2028 because the token has too many read/write sessions already open.
- 2029 • CKR_SESSION_EXISTS: This value can only be returned by **C_InitToken**. It indicates that a
 2030 session with the token is already open, and so the token cannot be initialized.
- 2031 • CKR_SESSION_PARALLEL_NOT_SUPPORTED: The specified token does not support parallel
 2032 sessions. This is a legacy error code—in Cryptoki Version 2.01 and up, *no* token supports parallel
 2033 sessions. CKR_SESSION_PARALLEL_NOT_SUPPORTED can only be returned by
 2034 **C_OpenSession**, and it is only returned when **C_OpenSession** is called in a particular [deprecated]
 2035 way.
- 2036 • CKR_SESSION_READ_ONLY: The specified session was unable to accomplish the desired action
 2037 because it is a read-only session. This return value has lower priority than
 2038 CKR_TOKEN_WRITE_PROTECTED.
- 2039 • CKR_SESSION_READ_ONLY_EXISTS: A read-only session already exists, and so the SO cannot
 2040 be logged in.
- 2041 • CKR_SESSION_READ_WRITE_SO_EXISTS: A read/write SO session already exists, and so a
 2042 read-only session cannot be opened.
- 2043 • CKR_SIGNATURE_LEN_RANGE: The provided signature/MAC can be seen to be invalid solely on
 2044 the basis of its length. This return value has higher priority than CKR_SIGNATURE_INVALID.
- 2045 • CKR_SIGNATURE_INVALID: The provided signature/MAC is invalid. This return value has lower
 2046 priority than CKR_SIGNATURE_LEN_RANGE.
- 2047 • CKR_SLOT_ID_INVALID: The specified slot ID is not valid.
- 2048 • CKR_STATE_UNSAVEABLE: The cryptographic operations state of the specified session cannot be
 2049 saved for some reason (possibly the token is simply unable to save the current state). This return
 2050 value has lower priority than CKR_OPERATION_NOT_INITIALIZED.
- 2051 • CKR_TEMPLATE_INCOMPLETE: The template specified for creating an object is incomplete, and
 2052 lacks some necessary attributes. See Section 4.1 for more information.
- 2053 • CKR_TEMPLATE_INCONSISTENT: The template specified for creating an object has conflicting
 2054 attributes. See Section 4.1 for more information.
- 2055 • CKR_TOKEN_NOT_RECOGNIZED: The Cryptoki library and/or slot does not recognize the token in
 2056 the slot.
- 2057 • CKR_TOKEN_WRITE_PROTECTED: The requested action could not be performed because the
 2058 token is write-protected. This return value has higher priority than CKR_SESSION_READ_ONLY.
- 2059 • CKR_UNWRAPPING_KEY_HANDLE_INVALID: This value can only be returned by **C_UnwrapKey**.
 2060 It indicates that the key handle specified to be used to unwrap another key is not valid.
- 2061 • CKR_UNWRAPPING_KEY_SIZE_RANGE: This value can only be returned by **C_UnwrapKey**. It
 2062 indicates that although the requested unwrapping operation could in principle be carried out, this
 2063 Cryptoki library (or the token) is unable to actually do it because the supplied key's size is outside the
 2064 range of key sizes that it can handle.
- 2065 • CKR_UNWRAPPING_KEY_TYPE_INCONSISTENT: This value can only be returned by
 2066 **C_UnwrapKey**. It indicates that the type of the key specified to unwrap another key is not consistent
 2067 with the mechanism specified for unwrapping.

- 2068 • CKR_USER_ALREADY_LOGGED_IN: This value can only be returned by **C_Login**. It indicates that
 2069 the specified user cannot be logged into the session, because it is already logged into the session.
 2070 For example, if an application has an open SO session, and it attempts to log the SO into it, it will
 2071 receive this error code.
- 2072 • CKR_USER_ANOTHER_ALREADY_LOGGED_IN: This value can only be returned by **C_Login**. It
 2073 indicates that the specified user cannot be logged into the session, because another user is already
 2074 logged into the session. For example, if an application has an open SO session, and it attempts to
 2075 log the normal user into it, it will receive this error code.
- 2076 • CKR_USER_NOT_LOGGED_IN: The desired action cannot be performed because the appropriate
 2077 user (or *an* appropriate user) is not logged in. One example is that a session cannot be logged out
 2078 unless it is logged in. Another example is that a private object cannot be created on a token unless
 2079 the session attempting to create it is logged in as the normal user. A final example is that
 2080 cryptographic operations on certain tokens cannot be performed unless the normal user is logged in.
- 2081 • CKR_USER_PIN_NOT_INITIALIZED: This value can only be returned by **C_Login**. It indicates that
 2082 the normal user's PIN has not yet been initialized with **C_InitPIN**.
- 2083 • CKR_USER_TOO_MANY_TYPES: An attempt was made to have more distinct users simultaneously
 2084 logged into the token than the token and/or library permits. For example, if some application has an
 2085 open SO session, and another application attempts to log the normal user into a session, the attempt
 2086 may return this error. It is not required to, however. Only if the simultaneous distinct users cannot be
 2087 supported does **C_Login** have to return this value. Note that this error code generalizes to true multi-
 2088 user tokens.
- 2089 • CKR_USER_TYPE_INVALID: An invalid value was specified as a **CK_USER_TYPE**. Valid types are
 2090 **CKU_SO**, **CKU_USER**, and **CKU_CONTEXT_SPECIFIC**.
- 2091 • CKR_WRAPPED_KEY_INVALID: This value can only be returned by **C_UnwrapKey**. It indicates
 2092 that the provided wrapped key is not valid. If a call is made to **C_UnwrapKey** to unwrap a particular
 2093 type of key (*i.e.*, some particular key type is specified in the template provided to **C_UnwrapKey**),
 2094 and the wrapped key provided to **C_UnwrapKey** is recognizably not a wrapped key of the proper
 2095 type, then **C_UnwrapKey** should return CKR_WRAPPED_KEY_INVALID. This return value has
 2096 lower priority than CKR_WRAPPED_KEY_LEN_RANGE.
- 2097 • CKR_WRAPPED_KEY_LEN_RANGE: This value can only be returned by **C_UnwrapKey**. It
 2098 indicates that the provided wrapped key can be seen to be invalid solely on the basis of its length.
 2099 This return value has higher priority than CKR_WRAPPED_KEY_INVALID.
- 2100 • CKR_WRAPPING_KEY_HANDLE_INVALID: This value can only be returned by **C_WrapKey**. It
 2101 indicates that the key handle specified to be used to wrap another key is not valid.
- 2102 • CKR_WRAPPING_KEY_SIZE_RANGE: This value can only be returned by **C_WrapKey**. It indicates
 2103 that although the requested wrapping operation could in principle be carried out, this Cryptoki library
 2104 (or the token) is unable to actually do it because the supplied wrapping key's size is outside the range
 2105 of key sizes that it can handle.
- 2106 • CKR_WRAPPING_KEY_TYPE_INCONSISTENT: This value can only be returned by **C_WrapKey**. It
 2107 indicates that the type of the key specified to wrap another key is not consistent with the mechanism
 2108 specified for wrapping.
- 2109 • CKR_OPERATION_CANCEL_FAILED: This value can only be returned by **C_SessionCancel**. It
 2110 means that one or more of the requested operations could not be cancelled for implementation or
 2111 vendor-specific reasons.

2112 5.1.7 More on relative priorities of Cryptoki errors

2113 In general, when a Cryptoki call is made, error codes from Section 5.1.1 (other than CKR_OK) take
 2114 precedence over error codes from Section 5.1.2, which take precedence over error codes from Section
 2115 5.1.3, which take precedence over error codes from Section 5.1.6. One minor implication of this is that
 2116 functions that use a session handle (*i.e.*, *most* functions!) never return the error code
 2117 CKR_TOKEN_NOT_PRESENT (they return CKR_SESSION_HANDLE_INVALID instead). Other than

2118 these precedences, if more than one error code applies to the result of a Cryptoki call, any of the
2119 applicable error codes may be returned. Exceptions to this rule will be explicitly mentioned in the
2120 descriptions of functions.

2121 5.1.8 Error code “gotchas”

2122 Here is a short list of a few particular things about return values that Cryptoki developers might want to be
2123 aware of:

- 2124 1. As mentioned in Sections 5.1.2 and 5.1.3, a Cryptoki library may not be able to make a distinction
2125 between a token being removed *before* a function invocation and a token being removed *during* a
2126 function invocation.
- 2127 2. As mentioned in Section 5.1.2, an application should never count on getting a
2128 CKR_SESSION_CLOSED error.
- 2129 3. The difference between CKR_DATA_INVALID and CKR_DATA_LEN_RANGE can be somewhat
2130 subtle. Unless an application *needs* to be able to distinguish between these return values, it is best to
2131 always treat them equivalently.
- 2132 4. Similarly, the difference between CKR_ENCRYPTED_DATA_INVALID and
2133 CKR_ENCRYPTED_DATA_LEN_RANGE, and between CKR_WRAPPED_KEY_INVALID and
2134 CKR_WRAPPED_KEY_LEN_RANGE, can be subtle, and it may be best to treat these return values
2135 equivalently.
- 2136 5. Even with the guidance of Section 4.1, it can be difficult for a Cryptoki library developer to know which
2137 of CKR_ATTRIBUTE_VALUE_INVALID, CKR_TEMPLATE_INCOMPLETE, or
2138 CKR_TEMPLATE_INCONSISTENT to return. When possible, it is recommended that application
2139 developers be generous in their interpretations of these error codes.

2140 5.2 Conventions for functions returning output in a variable-length 2141 buffer

2142 A number of the functions defined in Cryptoki return output produced by some cryptographic mechanism.
2143 The amount of output returned by these functions is returned in a variable-length application-supplied
2144 buffer. An example of a function of this sort is **C_Encrypt**, which takes some plaintext as an argument,
2145 and outputs a buffer full of ciphertext.

2146 These functions have some common calling conventions, which we describe here. Two of the arguments
2147 to the function are a pointer to the output buffer (say *pBuf*) and a pointer to a location which will hold the
2148 length of the output produced (say *pulBufLen*). There are two ways for an application to call such a
2149 function:

- 2150 1. If *pBuf* is NULL_PTR, then all that the function does is return (in **pulBufLen*) a number of bytes which
2151 would suffice to hold the cryptographic output produced from the input to the function. This number
2152 may somewhat exceed the precise number of bytes needed, but should not exceed it by a large
2153 amount. CKR_OK is returned by the function.
- 2154 2. If *pBuf* is not NULL_PTR, then **pulBufLen* MUST contain the size in bytes of the buffer pointed to by
2155 *pBuf*. If that buffer is large enough to hold the cryptographic output produced from the input to the
2156 function, then that cryptographic output is placed there, and CKR_OK is returned by the function. If
2157 the buffer is not large enough, then CKR_BUFFER_TOO_SMALL is returned. In either case,
2158 **pulBufLen* is set to hold the *exact* number of bytes needed to hold the cryptographic output produced
2159 from the input to the function.

2160 All functions which use the above convention will explicitly say so.

2161 Cryptographic functions which return output in a variable-length buffer should always return as much
2162 output as can be computed from what has been passed in to them thus far. As an example, consider a
2163 session which is performing a multiple-part decryption operation with DES in cipher-block chaining mode
2164 with PKCS padding. Suppose that, initially, 8 bytes of ciphertext are passed to the **C_DecryptUpdate**
2165 function. The block size of DES is 8 bytes, but the PKCS padding makes it unclear at this stage whether
2166 the ciphertext was produced from encrypting a 0-byte string, or from encrypting some string of length at

2167 least 8 bytes. Hence the call to **C_DecryptUpdate** should return 0 bytes of plaintext. If a single
2168 additional byte of ciphertext is supplied by a subsequent call to **C_DecryptUpdate**, then that call should
2169 return 8 bytes of plaintext (one full DES block).

2170 5.3 Disclaimer concerning sample code

2171 For the remainder of this section, we enumerate the various functions defined in Cryptoki. Most functions
2172 will be shown in use in at least one sample code snippet. For the sake of brevity, sample code will
2173 frequently be somewhat incomplete. In particular, sample code will generally ignore possible error
2174 returns from C library functions, and also will not deal with Cryptoki error returns in a realistic fashion.

2175 5.4 General-purpose functions

2176 Cryptoki provides the following general-purpose functions:

2177 5.4.1 C_Initialize

```
2178 CK_DECLARE_FUNCTION(CK_RV, C_Initialize) {  
2179     CK_VOID_PTR pInitArgs  
2180 };
```

2181 **C_Initialize** initializes the Cryptoki library. *pInitArgs* either has the value **NULL_PTR** or points to a
2182 **CK_C_INITIALIZE_ARGS** structure containing information on how the library should deal with multi-
2183 threaded access. If an application will not be accessing Cryptoki through multiple threads simultaneously,
2184 it can generally supply the value **NULL_PTR** to **C_Initialize** (the consequences of supplying this value will
2185 be explained below).

2186 If *pInitArgs* is non-**NULL_PTR**, **C_Initialize** should cast it to a **CK_C_INITIALIZE_ARGS_PTR** and then
2187 dereference the resulting pointer to obtain the **CK_C_INITIALIZE_ARGS** fields *CreateMutex*,
2188 *DestroyMutex*, *LockMutex*, *UnlockMutex*, *flags*, and *pReserved*. For this version of Cryptoki, the value of
2189 *pReserved* thereby obtained MUST be **NULL_PTR**; if it's not, then **C_Initialize** should return with the
2190 value **CKR_ARGUMENTS_BAD**.

2191 If the **CKF_LIBRARY_CANT_CREATE_OS_THREADS** flag in the *flags* field is set, that indicates that
2192 application threads which are executing calls to the Cryptoki library are not permitted to use the native
2193 operation system calls to spawn off new threads. In other words, the library's code may not create its
2194 own threads. If the library is unable to function properly under this restriction, **C_Initialize** should return
2195 with the value **CKR_NEED_TO_CREATE_THREADS**.

2196 A call to **C_Initialize** specifies one of four different ways to support multi-threaded access via the value of
2197 the **CKF_OS_LOCKING_OK** flag in the *flags* field and the values of the *CreateMutex*, *DestroyMutex*,
2198 *LockMutex*, and *UnlockMutex* function pointer fields:

- 2199 1. If the flag *isn't* set, and the function pointer fields *aren't* supplied (*i.e.*, they all have the value
2200 **NULL_PTR**), that means that the application *won't* be accessing the Cryptoki library from multiple
2201 threads simultaneously.
- 2202 2. If the flag *is* set, and the function pointer fields *aren't* supplied (*i.e.*, they all have the value
2203 **NULL_PTR**), that means that the application *will* be performing multi-threaded Cryptoki access, and
2204 the library needs to use the native operating system primitives to ensure safe multi-threaded access.
2205 If the library is unable to do this, **C_Initialize** should return with the value **CKR_CANT_LOCK**.
- 2206 3. If the flag *isn't* set, and the function pointer fields *are* supplied (*i.e.*, they all have non-**NULL_PTR**
2207 values), that means that the application *will* be performing multi-threaded Cryptoki access, and the
2208 library needs to use the supplied function pointers for mutex-handling to ensure safe multi-threaded
2209 access. If the library is unable to do this, **C_Initialize** should return with the value
2210 **CKR_CANT_LOCK**.
- 2211 4. If the flag *is* set, and the function pointer fields *are* supplied (*i.e.*, they all have non-**NULL_PTR**
2212 values), that means that the application *will* be performing multi-threaded Cryptoki access, and the
2213 library needs to use either the native operating system primitives or the supplied function pointers for

2214 mutex-handling to ensure safe multi-threaded access. If the library is unable to do this, **C_Initialize**
2215 should return with the value **CKR_CANT_LOCK**.

2216 If some, but not all, of the supplied function pointers to **C_Initialize** are non-NULL_PTR, then **C_Initialize**
2217 should return with the value **CKR_ARGUMENTS_BAD**.

2218 A call to **C_Initialize** with *pInitArgs* set to NULL_PTR is treated like a call to **C_Initialize** with *pInitArgs*
2219 pointing to a **CK_C_INITIALIZE_ARGS** which has the *CreateMutex*, *DestroyMutex*, *LockMutex*,
2220 *UnlockMutex*, and *pReserved* fields set to NULL_PTR, and has the *flags* field set to 0.

2221 **C_Initialize** should be the first Cryptoki call made by an application, except for calls to
2222 **C_GetFunctionList**, **C_GetInterfaceList**, or **C_GetInterface**. What this function actually does is
2223 implementation-dependent; typically, it might cause Cryptoki to initialize its internal memory buffers, or
2224 any other resources it requires.

2225 If several applications are using Cryptoki, each one should call **C_Initialize**. Every call to **C_Initialize**
2226 should (eventually) be succeeded by a single call to **C_Finalize**. See [\[PKCS11-UG\]](#) for further details.

2227 Return values: **CKR_ARGUMENTS_BAD**, **CKR_CANT_LOCK**,
2228 **CKR_CRYPTOKI_ALREADY_INITIALIZED**, **CKR_FUNCTION_FAILED**, **CKR_GENERAL_ERROR**,
2229 **CKR_HOST_MEMORY**, **CKR_NEED_TO_CREATE_THREADS**, **CKR_OK**.

2230 Example: see **C_GetInfo**.

2231 5.4.2 C_Finalize

```
2232 CK_DECLARE_FUNCTION(CK_RV, C_Finalize) (  
2233     CK_VOID_PTR pReserved  
2234 );
```

2235 **C_Finalize** is called to indicate that an application is finished with the Cryptoki library. It should be the
2236 last Cryptoki call made by an application. The *pReserved* parameter is reserved for future versions; for
2237 this version, it should be set to NULL_PTR (if **C_Finalize** is called with a non-NULL_PTR value for
2238 *pReserved*, it should return the value **CKR_ARGUMENTS_BAD**).

2239 If several applications are using Cryptoki, each one should call **C_Finalize**. Each application's call to
2240 **C_Finalize** should be preceded by a single call to **C_Initialize**; in between the two calls, an application
2241 can make calls to other Cryptoki functions. See [\[PKCS11-UG\]](#) for further details.

2242 *Despite the fact that the parameters supplied to **C_Initialize** can in general allow for safe multi-threaded*
2243 *access to a Cryptoki library, the behavior of **C_Finalize** is nevertheless undefined if it is called by an*
2244 *application while other threads of the application are making Cryptoki calls. The exception to this*
2245 *exceptional behavior of **C_Finalize** occurs when a thread calls **C_Finalize** while another of the*
2246 *application's threads is blocking on Cryptoki's **C_WaitForSlotEvent** function. When this happens, the*
2247 *blocked thread becomes unblocked and returns the value **CKR_CRYPTOKI_NOT_INITIALIZED**. See*
2248 ***C_WaitForSlotEvent** for more information.*

2249 Return values: **CKR_ARGUMENTS_BAD**, **CKR_CRYPTOKI_NOT_INITIALIZED**,
2250 **CKR_FUNCTION_FAILED**, **CKR_GENERAL_ERROR**, **CKR_HOST_MEMORY**, **CKR_OK**.

2251 Example: see **C_GetInfo**.

2252 5.4.3 C_GetInfo

```
2253 CK_DECLARE_FUNCTION(CK_RV, C_GetInfo) (  
2254     CK_INFO_PTR pInfo  
2255 );
```

2256 **C_GetInfo** returns general information about Cryptoki. *pInfo* points to the location that receives the
2257 information.

2258 Return values: **CKR_ARGUMENTS_BAD**, **CKR_CRYPTOKI_NOT_INITIALIZED**,
2259 **CKR_FUNCTION_FAILED**, **CKR_GENERAL_ERROR**, **CKR_HOST_MEMORY**, **CKR_OK**.

2260 Example:

```
2261 CK_INFO info;
2262 CK_RV rv;
2263 CK_C_INITIALIZE_ARGS InitArgs;
2264
2265 InitArgs.CreateMutex = &MyCreateMutex;
2266 InitArgs.DestroyMutex = &MyDestroyMutex;
2267 InitArgs.LockMutex = &MyLockMutex;
2268 InitArgs.UnlockMutex = &MyUnlockMutex;
2269 InitArgs.flags = CKF_OS_LOCKING_OK;
2270 InitArgs.pReserved = NULL_PTR;
2271
2272 rv = C_Initialize((CK_VOID_PTR)&InitArgs);
2273 assert(rv == CKR_OK);
2274
2275 rv = C_GetInfo(&info);
2276 assert(rv == CKR_OK);
2277 if(info.version.major == 2) {
2278     /* Do lots of interesting cryptographic things with the token */
2279     .
2280     .
2281 }
2282
2283 rv = C_Finalize(NULL_PTR);
2284 assert(rv == CKR_OK);
```

2285 5.4.4 C_GetFunctionList

```
2286 CK_DECLARE_FUNCTION(CK_RV, C_GetFunctionList)(
2287     CK_FUNCTION_LIST_PTR_PTR ppFunctionList
2288 );
```

2289 **C_GetFunctionList** obtains a pointer to the Cryptoki library's list of function pointers. *ppFunctionList*
2290 points to a value which will receive a pointer to the library's **CK_FUNCTION_LIST** structure, which in turn
2291 contains function pointers for all the Cryptoki API routines in the library. *The pointer thus obtained may*
2292 *point into memory which is owned by the Cryptoki library, and which may or may not be writable.*
2293 Whether or not this is the case, no attempt should be made to write to this memory.

2294 **C_GetFunctionList**, **C_GetInterfaceList**, and **C_GetInterface** are the only Cryptoki functions which an
2295 application may call before calling **C_Initialize**. It is provided to make it easier and faster for applications
2296 to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously.

2297 Return values: CKR_ARGUMENTS_BAD, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
2298 CKR_HOST_MEMORY, CKR_OK.

2299 Example:

```
2300 CK_FUNCTION_LIST_PTR pFunctionList;
2301 CK_C_Initialize pC_Initialize;
2302 CK_RV rv;
```



```

2303
2304 /* It's OK to call C_GetFunctionList before calling C_Initialize */
2305 rv = C_GetFunctionList(&pFunctionList);
2306 assert(rv == CKR_OK);
2307 pC_Initialize = pFunctionList -> C_Initialize;
2308
2309 /* Call the C_Initialize function in the library */
2310 rv = (*pC_Initialize)(NULL_PTR);

```

2311 5.4.5 C_GetInterfaceList

```

2312 CK_DECLARE_FUNCTION(CK_RV, C_GetInterfaceList)(
2313     CK_INTERFACE_PTR      pInterfaceList,
2314     CK_ULONG_PTR          pulCount
2315 );

```

2316 **C_GetInterfaceList** is used to obtain a list of interfaces supported by a Cryptoki library. *pulCount* points
2317 to the location that receives the number of interfaces.

2318 There are two ways for an application to call **C_GetInterfaceList**:

- 2319 1. If *pInterfaceList* is `NULL_PTR`, then all that **C_GetInterfaceList** does is return (in **pulCount*) the
2320 number of interfaces, without actually returning a list of interfaces. The contents of **pulCount* on
2321 entry to **C_GetInterfaceList** has no meaning in this case, and the call returns the value `CKR_OK`.
- 2322 2. If *pInterfaceList* is not `NULL_PTR`, then **pulCount* MUST contain the size (in terms of
2323 **CK_INTERFACE** elements) of the buffer pointed to by *pInterfaceList*. If that buffer is large enough to
2324 hold the list of interfaces, then the list is returned in it, and `CKR_OK` is returned. If not, then the call
2325 to **C_GetInterfaceList** returns the value `CKR_BUFFER_TOO_SMALL`. In either case, the value
2326 **pulCount* is set to hold the number of interfaces.

2327 Because **C_GetInterfaceList** does not allocate any space of its own, an application will often call
2328 **C_GetInterfaceList** twice. However, this behavior is by no means required.

2329 **C_GetInterfaceList** obtains (in **pFunctionList* of each interface) a pointer to the Cryptoki library's list of
2330 function pointers. *The pointer thus obtained may point into memory which is owned by the Cryptoki*
2331 *library, and which may or may not be writable.* Whether or not this is the case, no attempt should be
2332 made to write to this memory. The same caveat applies to the interface names returned.

2333
2334 **C_GetFunctionList**, **C_GetInterfaceList**, and **C_GetInterface** are the only Cryptoki functions which an
2335 application may call before calling **C_Initialize**. It is provided to make it easier and faster for applications
2336 to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously.

2337 Return values: `CKR_BUFFER_TOO_SMALL`, `CKR_ARGUMENTS_BAD`, `CKR_FUNCTION_FAILED`,
2338 `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`.

2339 Example:

```

2340 CK_ULONG ulCount=0;
2341 CK_INTERFACE_PTR interfaceList=NULL;
2342 CK_RV rv;
2343
2344 /* get number of interfaces */
2345 rv = C_GetInterfaceList(NULL,&ulCount);
2346 if (rv == CKR_OK) {
2347     /* get copy of interfaces */

```

```

2348     interfaceList = (CK_INTERFACE_PTR)malloc(ulCount*sizeof(CK_INTERFACE));
2349     rv = C_GetInterfaceList(interfaceList,&ulCount);
2350     for(i=0;i<ulCount;i++) {
2351         printf("interface %s version %d.%d funcs %p flags 0x%lu\n",
2352             interfaceList[i].pInterfaceName,
2353             ((CK_VERSION *)interfaceList[i].pFunctionList)->major,
2354             ((CK_VERSION *)interfaceList[i].pFunctionList)->minor,
2355             interfaceList[i].pFunctionList,
2356             interfaceList[i].flags);
2357     }
2358 }
2359

```

2360 5.4.6 C_GetInterface

```

2361 CK_DECLARE_FUNCTION(CK_RV,C_GetInterface) (
2362     CK_UTF8CHAR_PTR      pInterfaceName,
2363     CK_VERSION_PTR       pVersion,
2364     CK_INTERFACE_PTR_PTR ppInterface,
2365     CK_FLAGS              flags
2366 );

```

2367 **C_GetInterface** is used to obtain an interface supported by a Cryptoki library. *pInterfaceName* specifies the name of the interface, *pVersion* specifies the interface version, *ppInterface* points to the location that receives the interface, *flags* specifies the required interface flags.

2370 There are multiple ways for an application to specify a particular interface when calling **C_GetInterface**:

- 2371 1. If *pInterfaceName* is not NULL_PTR, the name of the interface returned must match. If
- 2372 *pInterfaceName* is NULL_PTR, the cryptoki library can return a default interface of its choice
- 2373 2. If *pVersion* is not NULL_PTR, the version of the interface returned must match. If *pVersion* is
- 2374 NULL_PTR, the cryptoki library can return an interface of any version
- 2375 3. If *flags* is non-zero, the interface returned must match all of the supplied flag values (but may include
- 2376 additional flags not specified). If *flags* is 0, the cryptoki library can return an interface with any flags

2377 **C_GetInterface** obtains (in **pFunctionList* of each interface) a pointer to the Cryptoki library's list of function pointers. *The pointer thus obtained may point into memory which is owned by the Cryptoki library, and which may or may not be writable.* Whether or not this is the case, no attempt should be made to write to this memory. The same caveat applies to the interface names returned.

2381 **C_GetFunctionList**, **C_GetInterfaceList**, and **C_GetInterface** are the only Cryptoki functions which an application may call before calling **C_Initialize**. It is provided to make it easier and faster for applications to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously.

2384 Return values: CKR_BUFFER_TOO_SMALL, CKR_ARGUMENTS_BAD, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK.

2386 Example:

```

2387 CK_INTERFACE_PTR interface;
2388 CK_RV rv;
2389 CK_VERSION version;
2390 CK_FLAGS flags=CKF_FORK_SAFE_INTERFACE;
2391

```

```

2392 /* get default interface */
2393 rv = C_GetInterface(NULL, NULL, &interface, flags);
2394 if (rv == CKR_OK) {
2395     printf("interface %s version %d.%d funcs %p flags 0x%lu\n",
2396         interface->pInterfaceName,
2397         ((CK_VERSION *)interface->pFunctionList)->major,
2398         ((CK_VERSION *)interface->pFunctionList)->minor,
2399         interface->pFunctionList,
2400         interface->flags);
2401 }
2402
2403 /* get default standard interface */
2404 rv = C_GetInterface((CK_UTF8CHAR_PTR)"PKCS 11", NULL, &interface, flags);
2405 if (rv == CKR_OK) {
2406     printf("interface %s version %d.%d funcs %p flags 0x%lu\n",
2407         interface->pInterfaceName,
2408         ((CK_VERSION *)interface->pFunctionList)->major,
2409         ((CK_VERSION *)interface->pFunctionList)->minor,
2410         interface->pFunctionList,
2411         interface->flags);
2412 }
2413
2414 /* get specific standard version interface */
2415 version.major=3;
2416 version.minor=0;
2417 rv = C_GetInterface((CK_UTF8CHAR_PTR)"PKCS 11", &version, &interface, flags);
2418 if (rv == CKR_OK) {
2419     CK_FUNCTION_LIST_3_0_PTR pkcs11=interface->pFunctionList;
2420
2421     /* ... use the new functions */
2422     pkcs11->C_LoginUser(hSession, userType, pPin, ulPinLen,
2423                         pUsername, ulUsernameLen);
2424 }
2425
2426 /* get specific vendor version interface */
2427 version.major=1;
2428 version.minor=0;
2429 rv = C_GetInterface((CK_UTF8CHAR_PTR)
2430                     "Vendor VendorName", &version, &interface, flags);
2431 if (rv == CKR_OK) {
2432     CK_FUNCTION_LIST_VENDOR_1_0_PTR pkcs11=interface->pFunctionList;
2433
2434     /* ... use vendor specific functions */

```



```

2435     pkcs11->C_VendorFunction1(param1,param2,param3);
2436 }
2437

```

5.5 Slot and token management functions

Cryptoki provides the following functions for slot and token management:

5.5.1 C_GetSlotList

```

2441 CK_DECLARE_FUNCTION(CK_RV, C_GetSlotList)(
2442     CK_BBOOL tokenPresent,
2443     CK_SLOT_ID_PTR pSlotList,
2444     CK_ULONG_PTR pulCount
2445 );

```

C_GetSlotList is used to obtain a list of slots in the system. *tokenPresent* indicates whether the list obtained includes only those slots with a token present (CK_TRUE), or all slots (CK_FALSE); *pulCount* points to the location that receives the number of slots.

There are two ways for an application to call **C_GetSlotList**:

1. If *pSlotList* is NULL_PTR, then all that **C_GetSlotList** does is return (in **pulCount*) the number of slots, without actually returning a list of slots. The contents of the buffer pointed to by *pulCount* on entry to **C_GetSlotList** has no meaning in this case, and the call returns the value CKR_OK.
2. If *pSlotList* is not NULL_PTR, then **pulCount* MUST contain the size (in terms of **CK_SLOT_ID** elements) of the buffer pointed to by *pSlotList*. If that buffer is large enough to hold the list of slots, then the list is returned in it, and CKR_OK is returned. If not, then the call to **C_GetSlotList** returns the value CKR_BUFFER_TOO_SMALL. In either case, the value **pulCount* is set to hold the number of slots.

Because **C_GetSlotList** does not allocate any space of its own, an application will often call **C_GetSlotList** twice (or sometimes even more times—if an application is trying to get a list of all slots with a token present, then the number of such slots can (unfortunately) change between when the application asks for how many such slots there are and when the application asks for the slots themselves). However, multiple calls to **C_GetSlotList** are by no means *required*.

All slots which **C_GetSlotList** reports MUST be able to be queried as valid slots by **C_GetSlotInfo**. Furthermore, the set of slots accessible through a Cryptoki library is checked at the time that **C_GetSlotList**, for list length prediction (NULL *pSlotList* argument) is called. If an application calls **C_GetSlotList** with a non-NULL *pSlotList*, and *then* the user adds or removes a hardware device, the changed slot list will only be visible and effective if **C_GetSlotList** is called again with NULL. Even if **C_GetSlotList** is successfully called this way, it may or may not be the case that the changed slot list will be successfully recognized depending on the library implementation. On some platforms, or earlier PKCS11 compliant libraries, it may be necessary to successfully call **C_Initialize** or to restart the entire system.

Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK.

Example:

```

2476 CK_ULONG ulSlotCount, ulSlotWithTokenCount;
2477 CK_SLOT_ID_PTR pSlotList, pSlotWithTokenList;
2478 CK_RV rv;
2479
2480 /* Get list of all slots */

```

```

2481 rv = C_GetSlotList(CK_FALSE, NULL_PTR, &ulSlotCount);
2482 if (rv == CKR_OK) {
2483     pSlotList =
2484         (CK_SLOT_ID_PTR) malloc(ulSlotCount*sizeof(CK_SLOT_ID));
2485     rv = C_GetSlotList(CK_FALSE, pSlotList, &ulSlotCount);
2486     if (rv == CKR_OK) {
2487         /* Now use that list of all slots */
2488         .
2489         .
2490     }
2491
2492     free(pSlotList);
2493 }
2494
2495 /* Get list of all slots with a token present */
2496 pSlotWithTokenList = (CK_SLOT_ID_PTR) malloc(0);
2497 ulSlotWithTokenCount = 0;
2498 while (1) {
2499     rv = C_GetSlotList(
2500         CK_TRUE, pSlotWithTokenList, ulSlotWithTokenCount);
2501     if (rv != CKR_BUFFER_TOO_SMALL)
2502         break;
2503     pSlotWithTokenList = realloc(
2504         pSlotWithTokenList,
2505         ulSlotWithTokenList*sizeof(CK_SLOT_ID));
2506 }
2507
2508 if (rv == CKR_OK) {
2509     /* Now use that list of all slots with a token present */
2510     .
2511     .
2512 }
2513
2514 free(pSlotWithTokenList);

```

5.5.2 C_GetSlotInfo

```

2516 CK_DECLARE_FUNCTION(CK_RV, C_GetSlotInfo) (
2517     CK_SLOT_ID slotID,
2518     CK_SLOT_INFO_PTR pInfo
2519 );

```

C_GetSlotInfo obtains information about a particular slot in the system. *slotID* is the ID of the slot; *pInfo* points to the location that receives the slot information.

2522 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
2523 CKR_DEVICE_ERROR, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY,
2524 CKR_OK, CKR_SLOT_ID_INVALID.

2525 Example: see **C_GetTokenInfo**.

2526 5.5.3 C_GetTokenInfo

```
2527 CK_DECLARE_FUNCTION(CK_RV, C_GetTokenInfo) (  
2528     CK_SLOT_ID slotID,  
2529     CK_TOKEN_INFO_PTR pInfo  
2530 );
```

2531 **C_GetTokenInfo** obtains information about a particular token in the system. *slotID* is the ID of the
2532 token's slot; *pInfo* points to the location that receives the token information.

2533 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
2534 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
2535 CKR_HOST_MEMORY, CKR_OK, CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT,
2536 CKR_TOKEN_NOT_RECOGNIZED, CKR_ARGUMENTS_BAD.

2537 Example:

```
2538 CK_ULONG ulCount;  
2539 CK_SLOT_ID_PTR pSlotList;  
2540 CK_SLOT_INFO slotInfo;  
2541 CK_TOKEN_INFO tokenInfo;  
2542 CK_RV rv;  
2543  
2544 rv = C_GetSlotList(CK_FALSE, NULL_PTR, &ulCount);  
2545 if ((rv == CKR_OK) && (ulCount > 0)) {  
2546     pSlotList = (CK_SLOT_ID_PTR) malloc(ulCount*sizeof(CK_SLOT_ID));  
2547     rv = C_GetSlotList(CK_FALSE, pSlotList, &ulCount);  
2548     assert(rv == CKR_OK);  
2549  
2550     /* Get slot information for first slot */  
2551     rv = C_GetSlotInfo(pSlotList[0], &slotInfo);  
2552     assert(rv == CKR_OK);  
2553  
2554     /* Get token information for first slot */  
2555     rv = C_GetTokenInfo(pSlotList[0], &tokenInfo);  
2556     if (rv == CKR_TOKEN_NOT_PRESENT) {  
2557         .  
2558         .  
2559     }  
2560     .  
2561     .  
2562     free(pSlotList);  
2563 }
```

5.5.4 C_WaitForSlotEvent

```
CK_DECLARE_FUNCTION(CK_RV, C_WaitForSlotEvent)(
    CK_FLAGS flags,
    CK_SLOT_ID_PTR pSlot,
    CK_VOID_PTR pReserved
);
```

C_WaitForSlotEvent waits for a slot event, such as token insertion or token removal, to occur. *flags* determines whether or not the **C_WaitForSlotEvent** call blocks (*i.e.*, waits for a slot event to occur); *pSlot* points to a location which will receive the ID of the slot that the event occurred in. *pReserved* is reserved for future versions; for this version of Cryptoki, it should be NULL_PTR.

At present, the only flag defined for use in the *flags* argument is **CKF_DONT_BLOCK**:

Internally, each Cryptoki application has a flag for each slot which is used to track whether or not any unrecognized events involving that slot have occurred. When an application initially calls **C_Initialize**, every slot's event flag is cleared. Whenever a slot event occurs, the flag corresponding to the slot in which the event occurred is set.

If **C_WaitForSlotEvent** is called with the **CKF_DONT_BLOCK** flag set in the *flags* argument, and some slot's event flag is set, then that event flag is cleared, and the call returns with the ID of that slot in the location pointed to by *pSlot*. If more than one slot's event flag is set at the time of the call, one such slot is chosen by the library to have its event flag cleared and to have its slot ID returned.

If **C_WaitForSlotEvent** is called with the **CKF_DONT_BLOCK** flag set in the *flags* argument, and no slot's event flag is set, then the call returns with the value CKR_NO_EVENT. In this case, the contents of the location pointed to by *pSlot* when **C_WaitForSlotEvent** are undefined.

If **C_WaitForSlotEvent** is called with the **CKF_DONT_BLOCK** flag clear in the *flags* argument, then the call behaves as above, except that it will block. That is, if no slot's event flag is set at the time of the call, **C_WaitForSlotEvent** will wait until some slot's event flag becomes set. If a thread of an application has a **C_WaitForSlotEvent** call blocking when another thread of that application calls **C_Finalize**, the **C_WaitForSlotEvent** call returns with the value CKR_CRYPTOKI_NOT_INITIALIZED.

Although the parameters supplied to C_Initialize can in general allow for safe multi-threaded access to a Cryptoki library, C_WaitForSlotEvent is exceptional in that the behavior of Cryptoki is undefined if multiple threads of a single application make simultaneous calls to C_WaitForSlotEvent.

Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_NO_EVENT, CKR_OK.

Example:

```
CK_FLAGS flags = 0;
CK_SLOT_ID slotID;
CK_SLOT_INFO slotInfo;

.
.

/* Block and wait for a slot event */
rv = C_WaitForSlotEvent(flags, &slotID, NULL_PTR);
assert(rv == CKR_OK);

/* See what's up with that slot */
rv = C_GetSlotInfo(slotID, &slotInfo);
assert(rv == CKR_OK);
```

2611

2612 5.5.5 C_GetMechanismList

```
2613 CK_DECLARE_FUNCTION(CK_RV, C_GetMechanismList) (  
2614     CK_SLOT_ID slotID,  
2615     CK_MECHANISM_TYPE_PTR pMechanismList,  
2616     CK_ULONG_PTR pulCount  
2617 );
```

2618 **C_GetMechanismList** is used to obtain a list of mechanism types supported by a token. *SlotID* is the ID
2619 of the token's slot; *pulCount* points to the location that receives the number of mechanisms.

2620 There are two ways for an application to call **C_GetMechanismList**:

- 2621 1. If *pMechanismList* is **NULL_PTR**, then all that **C_GetMechanismList** does is return (in **pulCount*)
2622 the number of mechanisms, without actually returning a list of mechanisms. The contents of
2623 **pulCount* on entry to **C_GetMechanismList** has no meaning in this case, and the call returns the
2624 value **CKR_OK**.
- 2625 2. If *pMechanismList* is not **NULL_PTR**, then **pulCount* **MUST** contain the size (in terms of
2626 **CK_MECHANISM_TYPE** elements) of the buffer pointed to by *pMechanismList*. If that buffer is large
2627 enough to hold the list of mechanisms, then the list is returned in it, and **CKR_OK** is returned. If not,
2628 then the call to **C_GetMechanismList** returns the value **CKR_BUFFER_TOO_SMALL**. In either
2629 case, the value **pulCount* is set to hold the number of mechanisms.

2630 Because **C_GetMechanismList** does not allocate any space of its own, an application will often call
2631 **C_GetMechanismList** twice. However, this behavior is by no means required.

2632 Return values: **CKR_BUFFER_TOO_SMALL**, **CKR_CRYPTOKI_NOT_INITIALIZED**,
2633 **CKR_DEVICE_ERROR**, **CKR_DEVICE_MEMORY**, **CKR_DEVICE_REMOVED**,
2634 **CKR_FUNCTION_FAILED**, **CKR_GENERAL_ERROR**, **CKR_HOST_MEMORY**, **CKR_OK**,
2635 **CKR_SLOT_ID_INVALID**, **CKR_TOKEN_NOT_PRESENT**, **CKR_TOKEN_NOT_RECOGNIZED**,
2636 **CKR_ARGUMENTS_BAD**.

2637 Example:

```
2638 CK_SLOT_ID slotID;  
2639 CK_ULONG ulCount;  
2640 CK_MECHANISM_TYPE_PTR pMechanismList;  
2641 CK_RV rv;  
2642  
2643 .  
2644 .  
2645 rv = C_GetMechanismList(slotID, NULL_PTR, &ulCount);  
2646 if ((rv == CKR_OK) && (ulCount > 0)) {  
2647     pMechanismList =  
2648         (CK_MECHANISM_TYPE_PTR)  
2649         malloc(ulCount*sizeof(CK_MECHANISM_TYPE));  
2650     rv = C_GetMechanismList(slotID, pMechanismList, &ulCount);  
2651     if (rv == CKR_OK) {  
2652         .  
2653         .  
2654     }  
2655     free(pMechanismList);
```

5.5.6 C_GetMechanismInfo

```
CK_DECLARE_FUNCTION(CK_RV, C_GetMechanismInfo) (
    CK_SLOT_ID slotID,
    CK_MECHANISM_TYPE type,
    CK_MECHANISM_INFO_PTR pInfo
);
```

C_GetMechanismInfo obtains information about a particular mechanism possibly supported by a token. *slotID* is the ID of the token's slot; *type* is the type of mechanism; *pInfo* points to the location that receives the mechanism information.

Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_MECHANISM_INVALID, CKR_OK, CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT, CKR_TOKEN_NOT_RECOGNIZED, CKR_ARGUMENTS_BAD.

Example:

```
CK_SLOT_ID slotID;
CK_MECHANISM_INFO info;
CK_RV rv;

.
.
/* Get information about the CKM_MD2 mechanism for this token */
rv = C_GetMechanismInfo(slotID, CKM_MD2, &info);
if (rv == CKR_OK) {
    if (info.flags & CKF_DIGEST) {
        .
        .
    }
}
```

5.5.7 C_InitToken

```
CK_DECLARE_FUNCTION(CK_RV, C_InitToken) (
    CK_SLOT_ID slotID,
    CK_UTF8CHAR_PTR pPin,
    CK_ULONG ulPinLen,
    CK_UTF8CHAR_PTR pLabel
);
```

C_InitToken initializes a token. *slotID* is the ID of the token's slot; *pPin* points to the SO's initial PIN (which need *not* be null-terminated); *ulPinLen* is the length in bytes of the PIN; *pLabel* points to the 32-byte label of the token (which **MUST** be padded with blank characters, and which **MUST** *not* be null-terminated). This standard allows PIN values to contain any valid UTF8 character, but the token may impose subset restrictions.

If the token has not been initialized (i.e. new from the factory), then the *pPin* parameter becomes the initial value of the SO PIN. If the token is being reinitialized, the *pPin* parameter is checked against the existing SO PIN to authorize the initialization operation. In both cases, the SO PIN is the value *pPin* after the function completes successfully. If the SO PIN is lost, then the card **MUST** be reinitialized using a

2700 mechanism outside the scope of this standard. The **CKF_TOKEN_INITIALIZED** flag in the
2701 **CK_TOKEN_INFO** structure indicates the action that will result from calling **C_InitToken**. If set, the token
2702 will be reinitialized, and the client **MUST** supply the existing SO password in *pPin*.

2703 When a token is initialized, all objects that can be destroyed are destroyed (*i.e.*, all except for
2704 “indestructible” objects such as keys built into the token). Also, access by the normal user is disabled
2705 until the SO sets the normal user’s PIN. Depending on the token, some “default” objects may be created,
2706 and attributes of some objects may be set to default values.

2707 If the token has a “protected authentication path”, as indicated by the
2708 **CKF_PROTECTED_AUTHENTICATION_PATH** flag in its **CK_TOKEN_INFO** being set, then that means
2709 that there is some way for a user to be authenticated to the token without having the application send a
2710 PIN through the Cryptoki library. One such possibility is that the user enters a PIN on a PINpad on the
2711 token itself, or on the slot device. To initialize a token with such a protected authentication path, the *pPin*
2712 parameter to **C_InitToken** should be **NULL_PTR**. During the execution of **C_InitToken**, the SO’s PIN will
2713 be entered through the protected authentication path.

2714 If the token has a protected authentication path other than a PINpad, then it is token-dependent whether
2715 or not **C_InitToken** can be used to initialize the token.

2716 A token cannot be initialized if Cryptoki detects that *any* application has an open session with it; when a
2717 call to **C_InitToken** is made under such circumstances, the call fails with error **CKR_SESSION_EXISTS**.
2718 Unfortunately, it may happen when **C_InitToken** is called that some other application *does* have an open
2719 session with the token, but Cryptoki cannot detect this, because it cannot detect anything about other
2720 applications using the token. If this is the case, then the consequences of the **C_InitToken** call are
2721 undefined.

2722 The **C_InitToken** function may not be sufficient to properly initialize complex tokens. In these situations,
2723 an initialization mechanism outside the scope of Cryptoki **MUST** be employed. The definition of “complex
2724 token” is product specific.

2725 Return values: **CKR_CRYPTOKI_NOT_INITIALIZED**, **CKR_DEVICE_ERROR**, **CKR_DEVICE_MEMORY**,
2726 **CKR_DEVICE_REMOVED**, **CKR_FUNCTION_CANCELED**, **CKR_FUNCTION_FAILED**,
2727 **CKR_GENERAL_ERROR**, **CKR_HOST_MEMORY**, **CKR_OK**, **CKR_PIN_INCORRECT**,
2728 **CKR_PIN_LOCKED**, **CKR_SESSION_EXISTS**, **CKR_SLOT_ID_INVALID**,
2729 **CKR_TOKEN_NOT_PRESENT**, **CKR_TOKEN_NOT_RECOGNIZED**,
2730 **CKR_TOKEN_WRITE_PROTECTED**, **CKR_ARGUMENTS_BAD**.

2731 Example:

```
2732 CK_SLOT_ID slotID;  
2733 CK_UTF8CHAR_PTR pin = "MyPIN";  
2734 CK_UTF8CHAR label[32];  
2735 CK_RV rv;  
2736  
2737 .  
2738 .  
2739 memset(label, '\0', sizeof(label));  
2740 memcpy(label, "My first token", strlen("My first token"));  
2741 rv = C_InitToken(slotID, pin, strlen(pin), label);  
2742 if (rv == CKR_OK) {  
2743     .  
2744     .  
2745 }
```

5.5.8 C_InitPIN

```
CK_DECLARE_FUNCTION(CK_RV, C_InitPIN) (
    CK_SESSION_HANDLE hSession,
    CK_UTF8CHAR_PTR pPin,
    CK_ULONG ulPinLen
);
```

C_InitPIN initializes the normal user's PIN. *hSession* is the session's handle; *pPin* points to the normal user's PIN; *ulPinLen* is the length in bytes of the PIN. This standard allows PIN values to contain any valid UTF8 character, but the token may impose subset restrictions.

C_InitPIN can only be called in the "R/W SO Functions" state. An attempt to call it from a session in any other state fails with error CKR_USER_NOT_LOGGED_IN.

If the token has a "protected authentication path", as indicated by the CKF_PROTECTED_AUTHENTICATION_PATH flag in its **CK_TOKEN_INFO** being set, then that means that there is some way for a user to be authenticated to the token without having to send a PIN through the Cryptoki library. One such possibility is that the user enters a PIN on a PIN pad on the token itself, or on the slot device. To initialize the normal user's PIN on a token with such a protected authentication path, the *pPin* parameter to **C_InitPIN** should be NULL_PTR. During the execution of **C_InitPIN**, the SO will enter the new PIN through the protected authentication path.

If the token has a protected authentication path other than a PIN pad, then it is token-dependent whether or not **C_InitPIN** can be used to initialize the normal user's token access.

Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_PIN_INVALID, CKR_PIN_LEN_RANGE, CKR_SESSION_CLOSED, CKR_SESSION_READ_ONLY, CKR_SESSION_HANDLE_INVALID, CKR_TOKEN_WRITE_PROTECTED, CKR_USER_NOT_LOGGED_IN, CKR_ARGUMENTS_BAD.

Example:

```
CK_SESSION_HANDLE hSession;
CK_UTF8CHAR newPin[] = {"NewPIN"};
CK_RV rv;

rv = C_InitPIN(hSession, newPin, sizeof(newPin)-1);
if (rv == CKR_OK) {
    .
    .
}
```

5.5.9 C_SetPIN

```
CK_DECLARE_FUNCTION(CK_RV, C_SetPIN) (
    CK_SESSION_HANDLE hSession,
    CK_UTF8CHAR_PTR pOldPin,
    CK_ULONG ulOldLen,
    CK_UTF8CHAR_PTR pNewPin,
    CK_ULONG ulNewLen
);
```

C_SetPIN modifies the PIN of the user that is currently logged in, or the CKU_USER PIN if the session is not logged in. *hSession* is the session's handle; *pOldPin* points to the old PIN; *ulOldLen* is the length in bytes of the old PIN; *pNewPin* points to the new PIN; *ulNewLen* is the length in bytes of the new PIN. This

2793 standard allows PIN values to contain any valid UTF8 character, but the token may impose subset
 2794 restrictions.

2795 **C_SetPIN** can only be called in the “R/W Public Session” state, “R/W SO Functions” state, or “R/W User
 2796 Functions” state. An attempt to call it from a session in any other state fails with error
 2797 CKR_SESSION_READ_ONLY.

2798 If the token has a “protected authentication path”, as indicated by the
 2799 CKF_PROTECTED_AUTHENTICATION_PATH flag in its **CK_TOKEN_INFO** being set, then that means
 2800 that there is some way for a user to be authenticated to the token without having to send a PIN through
 2801 the Cryptoki library. One such possibility is that the user enters a PIN on a PIN pad on the token itself, or
 2802 on the slot device. To modify the current user’s PIN on a token with such a protected authentication path,
 2803 the *pOldPin* and *pNewPin* parameters to **C_SetPIN** should be NULL_PTR. During the execution of
 2804 **C_SetPIN**, the current user will enter the old PIN and the new PIN through the protected authentication
 2805 path. It is not specified how the PIN pad should be used to enter *two* PINs; this varies.

2806 If the token has a protected authentication path other than a PIN pad, then it is token-dependent whether
 2807 or not **C_SetPIN** can be used to modify the current user’s PIN.

2808 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
 2809 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
 2810 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_PIN_INCORRECT,
 2811 CKR_PIN_INVALID, CKR_PIN_LEN_RANGE, CKR_PIN_LOCKED, CKR_SESSION_CLOSED,
 2812 CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY,
 2813 CKR_TOKEN_WRITE_PROTECTED, CKR_ARGUMENTS_BAD.

2814 Example:

```

2815 CK_SESSION_HANDLE hSession;
2816 CK_UTF8CHAR oldPin[] = {"OldPIN"};
2817 CK_UTF8CHAR newPin[] = {"NewPIN"};
2818 CK_RV rv;
2819
2820 rv = C_SetPIN(
2821     hSession, oldPin, sizeof(oldPin)-1, newPin, sizeof(newPin)-1);
2822 if (rv == CKR_OK) {
2823     .
2824     .
2825 }
```

2826 5.6 Session management functions

2827 A typical application might perform the following series of steps to make use of a token (note that there
 2828 are other reasonable sequences of events that an application might perform):

- 2829 1. Select a token.
- 2830 2. Make one or more calls to **C_OpenSession** to obtain one or more sessions with the token.
- 2831 3. Call **C_Login** to log the user into the token. Since all sessions an application has with a token have a
 2832 shared login state, **C_Login** only needs to be called for one of the sessions.
- 2833 4. Perform cryptographic operations using the sessions with the token.
- 2834 5. Call **C_CloseSession** once for each session that the application has with the token, or call
 2835 **C_CloseAllSessions** to close all the application’s sessions simultaneously.

2836 As has been observed, an application may have concurrent sessions with more than one token. It is also
 2837 possible for a token to have concurrent sessions with more than one application.

2838 Cryptoki provides the following functions for session management:

5.6.1 C_OpenSession

```
CK_DECLARE_FUNCTION(CK_RV, C_OpenSession)(
    CK_SLOT_ID slotID,
    CK_FLAGS flags,
    CK_VOID_PTR pApplication,
    CK_NOTIFY Notify,
    CK_SESSION_HANDLE_PTR phSession
);
```

C_OpenSession opens a session between an application and a token in a particular slot. *slotID* is the slot's ID; *flags* indicates the type of session; *pApplication* is an application-defined pointer to be passed to the notification callback; *Notify* is the address of the notification callback function (see Section 5.21); *phSession* points to the location that receives the handle for the new session.

When opening a session with **C_OpenSession**, the *flags* parameter consists of the logical OR of zero or more bit flags defined in the **CK_SESSION_INFO** data type. For legacy reasons, the **CKF_SERIAL_SESSION** bit MUST always be set; if a call to **C_OpenSession** does not have this bit set, the call should return unsuccessfully with the error code **CKR_SESSION_PARALLEL_NOT_SUPPORTED**.

There may be a limit on the number of concurrent sessions an application may have with the token, which may depend on whether the session is "read-only" or "read/write". An attempt to open a session which does not succeed because there are too many existing sessions of some type should return **CKR_SESSION_COUNT**.

If the token is write-protected (as indicated in the **CK_TOKEN_INFO** structure), then only read-only sessions may be opened with it.

If the application calling **C_OpenSession** already has a R/W SO session open with the token, then any attempt to open a R/O session with the token fails with error code **CKR_SESSION_READ_WRITE_SO_EXISTS** (see [PKCS11-UG] for further details).

The *Notify* callback function is used by Cryptoki to notify the application of certain events. If the application does not wish to support callbacks, it should pass a value of **NULL_PTR** as the *Notify* parameter. See Section 5.21 for more information about application callbacks.

Return values: **CKR_CRYPTOKI_NOT_INITIALIZED**, **CKR_DEVICE_ERROR**, **CKR_DEVICE_MEMORY**, **CKR_DEVICE_REMOVED**, **CKR_FUNCTION_FAILED**, **CKR_GENERAL_ERROR**, **CKR_HOST_MEMORY**, **CKR_OK**, **CKR_SESSION_COUNT**, **CKR_SESSION_PARALLEL_NOT_SUPPORTED**, **CKR_SESSION_READ_WRITE_SO_EXISTS**, **CKR_SLOT_ID_INVALID**, **CKR_TOKEN_NOT_PRESENT**, **CKR_TOKEN_NOT_RECOGNIZED**, **CKR_TOKEN_WRITE_PROTECTED**, **CKR_ARGUMENTS_BAD**.

Example: see **C_CloseSession**.

5.6.2 C_CloseSession

```
CK_DECLARE_FUNCTION(CK_RV, C_CloseSession)(
    CK_SESSION_HANDLE hSession
);
```

C_CloseSession closes a session between an application and a token. *hSession* is the session's handle.

When a session is closed, all session objects created by the session are destroyed automatically, even if the application has other sessions "using" the objects (see [PKCS11-UG] for further details).

If this function is successful and it closes the last session between the application and the token, the login state of the token for the application returns to public sessions. Any new sessions to the token opened by the application will be either R/O Public or R/W Public sessions.

Depending on the token, when the last open session any application has with the token is closed, the token may be "ejected" from its reader (if this capability exists).

2888 Despite the fact this **C_CloseSession** is supposed to close a session, the return value
2889 CKR_SESSION_CLOSED is an *error* return. It actually indicates the (probably somewhat unlikely) event
2890 that while this function call was executing, another call was made to **C_CloseSession** to close this
2891 particular session, and that call finished executing first. Such uses of sessions are a bad idea, and
2892 Cryptoki makes little promise of what will occur in general if an application indulges in this sort of
2893 behavior.

2894 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
2895 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
2896 CKR_HOST_MEMORY, CKR_OK, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

2897 Example:

```
2898 CK_SLOT_ID slotID;  
2899 CK_BYTE application;  
2900 CK_NOTIFY MyNotify;  
2901 CK_SESSION_HANDLE hSession;  
2902 CK_RV rv;  
2903  
2904 .  
2905 .  
2906 application = 17;  
2907 MyNotify = &EncryptionSessionCallback;  
2908 rv = C_OpenSession(  
2909     slotID, CKF_SERIAL_SESSION | CKF_RW_SESSION,  
2910     (CK_VOID_PTR) &application, MyNotify,  
2911     &hSession);  
2912 if (rv == CKR_OK) {  
2913     .  
2914     .  
2915     C_CloseSession(hSession);  
2916 }
```

2917 5.6.3 C_CloseAllSessions

```
2918 CK_DECLARE_FUNCTION(CK_RV, C_CloseAllSessions) (  
2919     CK_SLOT_ID slotID  
2920 );
```

2921 **C_CloseAllSessions** closes all sessions an application has with a token. *slotID* specifies the token's slot.
2922 When a session is closed, all session objects created by the session are destroyed automatically.
2923 After successful execution of this function, the login state of the token for the application returns to public
2924 sessions. Any new sessions to the token opened by the application will be either R/O Public or R/W
2925 Public sessions.

2926 Depending on the token, when the last open session any application has with the token is closed, the
2927 token may be "ejected" from its reader (if this capability exists).

2928 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
2929 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
2930 CKR_HOST_MEMORY, CKR_OK, CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT.

2931 Example:

```
2932 CK_SLOT_ID slotID;
```

```

2933 CK_RV rv;
2934
2935 .
2936 .
2937 rv = C_CloseAllSessions(slotID);

```

2938 5.6.4 C_GetSessionInfo

```

2939 CK_DECLARE_FUNCTION(CK_RV, C_GetSessionInfo) (
2940     CK_SESSION_HANDLE hSession,
2941     CK_SESSION_INFO_PTR pInfo
2942 );

```

2943 **C_GetSessionInfo** obtains information about a session. *hSession* is the session's handle; *pInfo* points to
 2944 the location that receives the session information.

2945 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
 2946 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
 2947 CKR_HOST_MEMORY, CKR_OK, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
 2948 CKR_ARGUMENTS_BAD.

2949 Example:

```

2950 CK_SESSION_HANDLE hSession;
2951 CK_SESSION_INFO info;
2952 CK_RV rv;
2953
2954 .
2955 .
2956 rv = C_GetSessionInfo(hSession, &info);
2957 if (rv == CKR_OK) {
2958     if (info.state == CKS_RW_USER_FUNCTIONS) {
2959         .
2960         .
2961     }
2962     .
2963     .
2964 }

```

2965 5.6.5 C_SessionCancel

```

2966 CK_DECLARE_FUNCTION(CK_RV, C_SessionCancel) (
2967     CK_SESSION_HANDLE hSession
2968     CK_FLAGS flags
2969 );

```

2970 **C_SessionCancel** terminates active session based operations. *hSession* is the session's handle; *flags*
 2971 indicates the operations to cancel.

2972 To identify which operation(s) should be terminated, the *flags* parameter should be assigned the logical
 2973 bitwise OR of one or more of the bit flags defined in the **CK_MECHANISM_INFO** structure.

2974 If no flags are set, the session state will not be modified and CKR_OK will be returned.

2975 If a flag is set for an operation that has not been initialized in the session, the operation flag will be
 2976 ignored and **C_SessionCancel** will behave as if the operation flag was not set.

2977 If any of the operations indicated by the *flags* parameter cannot be cancelled,
2978 CKR_OPERATION_CANCEL_FAILED must be returned. If multiple operation flags were set and
2979 CKR_OPERATION_CANCEL_FAILED is returned, this function does not provide any information about
2980 which operation(s) could not be cancelled. If an application desires to know if any single operation could
2981 not be cancelled, the application should not call **C_SessionCancel** with multiple flags set.

2982 If **C_SessionCancel** is called from an application callback (see Section 5.16), no action will be taken by
2983 the library and CKR_FUNCTION_FAILED must be returned.

2984 If **C_SessionCancel** is used to cancel one half of a dual-function operation, the remaining operation
2985 should still be left in an active state. However, it is expected that some Cryptoki implementations may not
2986 support this and return CKR_OPERATION_CANCEL_FAILED unless flags for both operations are
2987 provided.

2988

2989 Example:

```
2990 CK_SESSION_HANDLE hSession;  
2991 CK_RV rv;  
2992  
2993 rv = C_EncryptInit(hSession, &mechanism, hKey);  
2994 if (rv != CKR_OK)  
2995 {  
2996     .  
2997     .  
2998 }  
2999  
3000 rv = C_SessionCancel (hSession, CKF_ENCRYPT);  
3001 if (rv != CKR_OK)  
3002 {  
3003     .  
3004     .  
3005 }  
3006  
3007 rv = C_EncryptInit(hSession, &mechanism, hKey);  
3008 if (rv != CKR_OK)  
3009 {  
3010     .  
3011     .  
3012 }  
3013
```

3014

3015

3016

3017 Below are modifications to existing API descriptions to allow an alternate method of cancelling individual
3018 operations. The additional text is highlighted.

3019 5.6.6 C_GetOperationState

```
3020 CK_DECLARE_FUNCTION(CK_RV, C_GetOperationState) (  
3021     CK_SESSION_HANDLE hSession,
```

```

3022     CK_BYTE_PTR pOperationState,
3023     CK_ULONG_PTR pulOperationStateLen
3024 );

```

3025 **C_GetOperationState** obtains a copy of the cryptographic operations state of a session, encoded as a
3026 string of bytes. *hSession* is the session's handle; *pOperationState* points to the location that receives the
3027 state; *pulOperationStateLen* points to the location that receives the length in bytes of the state.

3028 Although the saved state output by **C_GetOperationState** is not really produced by a "cryptographic
3029 mechanism", **C_GetOperationState** nonetheless uses the convention described in Section 5.2 on
3030 producing output.

3031 Precisely what the "cryptographic operations state" this function saves is varies from token to token;
3032 however, this state is what is provided as input to **C_SetOperationState** to restore the cryptographic
3033 activities of a session.

3034 Consider a session which is performing a message digest operation using SHA-1 (*i.e.*, the session is
3035 using the **CKM_SHA_1** mechanism). Suppose that the message digest operation was initialized
3036 properly, and that precisely 80 bytes of data have been supplied so far as input to SHA-1. The
3037 application now wants to "save the state" of this digest operation, so that it can continue it later. In this
3038 particular case, since SHA-1 processes 512 bits (64 bytes) of input at a time, the cryptographic
3039 operations state of the session most likely consists of three distinct parts: the state of SHA-1's 160-bit
3040 internal chaining variable; the 16 bytes of unprocessed input data; and some administrative data
3041 indicating that this saved state comes from a session which was performing SHA-1 hashing. Taken
3042 together, these three pieces of information suffice to continue the current hashing operation at a later
3043 time.

3044 Consider next a session which is performing an encryption operation with DES (a block cipher with a
3045 block size of 64 bits) in CBC (cipher-block chaining) mode (*i.e.*, the session is using the **CKM_DES_CBC**
3046 mechanism). Suppose that precisely 22 bytes of data (in addition to an IV for the CBC mode) have been
3047 supplied so far as input to DES, which means that the first two 8-byte blocks of ciphertext have already
3048 been produced and output. In this case, the cryptographic operations state of the session most likely
3049 consists of three or four distinct parts: the second 8-byte block of ciphertext (this will be used for cipher-
3050 block chaining to produce the next block of ciphertext); the 6 bytes of data still awaiting encryption; some
3051 administrative data indicating that this saved state comes from a session which was performing DES
3052 encryption in CBC mode; and possibly the DES key being used for encryption (see **C_SetOperationState**
3053 for more information on whether or not the key is present in the saved state).

3054 If a session is performing two cryptographic operations simultaneously (see Section 5.14), then the
3055 cryptographic operations state of the session will contain all the necessary information to restore both
3056 operations.

3057 An attempt to save the cryptographic operations state of a session which does not currently have some
3058 active savable cryptographic operation(s) (encryption, decryption, digesting, signing without message
3059 recovery, verification without message recovery, or some legal combination of two of these) should fail
3060 with the error **CKR_OPERATION_NOT_INITIALIZED**.

3061 An attempt to save the cryptographic operations state of a session which is performing an appropriate
3062 cryptographic operation (or two), but which cannot be satisfied for any of various reasons (certain
3063 necessary state information and/or key information can't leave the token, for example) should fail with the
3064 error **CKR_STATE_UNSAVEABLE**.

3065 Return values: **CKR_BUFFER_TOO_SMALL**, **CKR_CRYPTOKI_NOT_INITIALIZED**,
3066 **CKR_DEVICE_ERROR**, **CKR_DEVICE_MEMORY**, **CKR_DEVICE_REMOVED**,
3067 **CKR_FUNCTION_FAILED**, **CKR_GENERAL_ERROR**, **CKR_HOST_MEMORY**, **CKR_OK**,
3068 **CKR_OPERATION_NOT_INITIALIZED**, **CKR_SESSION_CLOSED**, **CKR_SESSION_HANDLE_INVALID**,
3069 **CKR_STATE_UNSAVEABLE**, **CKR_ARGUMENTS_BAD**.

3070 Example: see **C_SetOperationState**.

5.6.7 C_SetOperationState

```
CK_DECLARE_FUNCTION(CK_RV, C_SetOperationState)(  
    CK_SESSION_HANDLE hSession,  
    CK_BYTE_PTR pOperationState,  
    CK_ULONG ulOperationStateLen,  
    CK_OBJECT_HANDLE hEncryptionKey,  
    CK_OBJECT_HANDLE hAuthenticationKey  
);
```

C_SetOperationState restores the cryptographic operations state of a session from a string of bytes obtained with **C_GetOperationState**. *hSession* is the session's handle; *pOperationState* points to the location holding the saved state; *ulOperationStateLen* holds the length of the saved state; *hEncryptionKey* holds a handle to the key which will be used for an ongoing encryption or decryption operation in the restored session (or 0 if no encryption or decryption key is needed, either because no such operation is ongoing in the stored session or because all the necessary key information is present in the saved state); *hAuthenticationKey* holds a handle to the key which will be used for an ongoing signature, MACing, or verification operation in the restored session (or 0 if no such key is needed, either because no such operation is ongoing in the stored session or because all the necessary key information is present in the saved state).

The state need not have been obtained from the same session (the "source session") as it is being restored to (the "destination session"). However, the source session and destination session should have a common session state (e.g., CKS_RW_USER_FUNCTIONS), and should be with a common token. There is also no guarantee that cryptographic operations state may be carried across logins, or across different Cryptoki implementations.

If **C_SetOperationState** is supplied with alleged saved cryptographic operations state which it can determine is not valid saved state (or is cryptographic operations state from a session with a different session state, or is cryptographic operations state from a different token), it fails with the error CKR_SAVED_STATE_INVALID.

Saved state obtained from calls to **C_GetOperationState** may or may not contain information about keys in use for ongoing cryptographic operations. If a saved cryptographic operations state has an ongoing encryption or decryption operation, and the key in use for the operation is not saved in the state, then it MUST be supplied to **C_SetOperationState** in the *hEncryptionKey* argument. If it is not, then **C_SetOperationState** will fail and return the error CKR_KEY_NEEDED. If the key in use for the operation is saved in the state, then it *can* be supplied in the *hEncryptionKey* argument, but this is not required.

Similarly, if a saved cryptographic operations state has an ongoing signature, MACing, or verification operation, and the key in use for the operation is not saved in the state, then it MUST be supplied to **C_SetOperationState** in the *hAuthenticationKey* argument. If it is not, then **C_SetOperationState** will fail with the error CKR_KEY_NEEDED. If the key in use for the operation is saved in the state, then it *can* be supplied in the *hAuthenticationKey* argument, but this is not required.

If an *irrelevant* key is supplied to **C_SetOperationState** call (e.g., a nonzero key handle is submitted in the *hEncryptionKey* argument, but the saved cryptographic operations state supplied does not have an ongoing encryption or decryption operation, then **C_SetOperationState** fails with the error CKR_KEY_NOT_NEEDED.

If a key is supplied as an argument to **C_SetOperationState**, and **C_SetOperationState** can somehow detect that this key was not the key being used in the source session for the supplied cryptographic operations state (it may be able to detect this if the key or a hash of the key is present in the saved state, for example), then **C_SetOperationState** fails with the error CKR_KEY_CHANGED.

An application can look at the **CKF_RESTORE_KEY_NOT_NEEDED** flag in the flags field of the **CK_TOKEN_INFO** field for a token to determine whether or not it needs to supply key handles to **C_SetOperationState** calls. If this flag is true, then a call to **C_SetOperationState** *never* needs a key handle to be supplied to it. If this flag is false, then at least some of the time, **C_SetOperationState** requires a key handle, and so the application should probably *always* pass in any relevant key handles when restoring cryptographic operations state to a session.

3124 **C_SetOperationState** can successfully restore cryptographic operations state to a session even if that
3125 session has active cryptographic or object search operations when **C_SetOperationState** is called (the
3126 ongoing operations are abruptly cancelled).

3127 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
3128 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
3129 CKR_HOST_MEMORY, CKR_KEY_CHANGED, CKR_KEY_NEEDED, CKR_KEY_NOT_NEEDED,
3130 CKR_OK, CKR_SAVED_STATE_INVALID, CKR_SESSION_CLOSED,
3131 CKR_SESSION_HANDLE_INVALID, CKR_ARGUMENTS_BAD.

3132 Example:

```
3133 CK_SESSION_HANDLE hSession;  
3134 CK_MECHANISM digestMechanism;  
3135 CK_ULONG ulStateLen;  
3136 CK_BYTE data1[] = {0x01, 0x03, 0x05, 0x07};  
3137 CK_BYTE data2[] = {0x02, 0x04, 0x08};  
3138 CK_BYTE data3[] = {0x10, 0x0F, 0x0E, 0x0D, 0x0C};  
3139 CK_BYTE pDigest[20];  
3140 CK_ULONG ulDigestLen;  
3141 CK_RV rv;  
3142  
3143 .  
3144 .  
3145 /* Initialize hash operation */  
3146 rv = C_DigestInit(hSession, &digestMechanism);  
3147 assert(rv == CKR_OK);  
3148  
3149 /* Start hashing */  
3150 rv = C_DigestUpdate(hSession, data1, sizeof(data1));  
3151 assert(rv == CKR_OK);  
3152  
3153 /* Find out how big the state might be */  
3154 rv = C_GetOperationState(hSession, NULL_PTR, &ulStateLen);  
3155 assert(rv == CKR_OK);  
3156  
3157 /* Allocate some memory and then get the state */  
3158 pState = (CK_BYTE_PTR) malloc(ulStateLen);  
3159 rv = C_GetOperationState(hSession, pState, &ulStateLen);  
3160  
3161 /* Continue hashing */  
3162 rv = C_DigestUpdate(hSession, data2, sizeof(data2));  
3163 assert(rv == CKR_OK);  
3164  
3165 /* Restore state. No key handles needed */  
3166 rv = C_SetOperationState(hSession, pState, ulStateLen, 0, 0);  
3167 assert(rv == CKR_OK);
```



```

3168
3169 /* Continue hashing from where we saved state */
3170 rv = C_DigestUpdate(hSession, data3, sizeof(data3));
3171 assert(rv == CKR_OK);
3172
3173 /* Conclude hashing operation */
3174 ulDigestLen = sizeof(pDigest);
3175 rv = C_DigestFinal(hSession, pDigest, &ulDigestLen);
3176 if (rv == CKR_OK) {
3177     /* pDigest[] now contains the hash of 0x01030507100F0E0D0C */
3178     .
3179     .
3180 }

```

3181 5.6.8 C_Login

```

3182 CK_DECLARE_FUNCTION(CK_RV, C_Login) (
3183     CK_SESSION_HANDLE hSession,
3184     CK_USER_TYPE userType,
3185     CK_UTF8CHAR_PTR pPin,
3186     CK_ULONG ulPinLen
3187 );

```

3188 **C_Login** logs a user into a token. *hSession* is a session handle; *userType* is the user type; *pPin* points to
3189 the user's PIN; *ulPinLen* is the length of the PIN. This standard allows PIN values to contain any valid
3190 UTF8 character, but the token may impose subset restrictions.

3191 When the user type is either CKU_SO or CKU_USER, if the call succeeds, each of the application's
3192 sessions will enter either the "R/W SO Functions" state, the "R/W User Functions" state, or the "R/O User
3193 Functions" state. If the user type is CKU_CONTEXT_SPECIFIC, the behavior of C_Login depends on
3194 the context in which it is called. Improper use of this user type will result in a return value
3195 CKR_OPERATION_NOT_INITIALIZED..

3196 If the token has a "protected authentication path", as indicated by the
3197 **CKF_PROTECTED_AUTHENTICATION_PATH** flag in its **CK_TOKEN_INFO** being set, then that means
3198 that there is some way for a user to be authenticated to the token without having to send a PIN through
3199 the Cryptoki library. One such possibility is that the user enters a PIN on a PIN pad on the token itself, or
3200 on the slot device. Or the user might not even use a PIN—authentication could be achieved by some
3201 fingerprint-reading device, for example. To log into a token with a protected authentication path, the *pPin*
3202 parameter to **C_Login** should be NULL_PTR. When **C_Login** returns, whatever authentication method
3203 supported by the token will have been performed; a return value of CKR_OK means that the user was
3204 successfully authenticated, and a return value of CKR_PIN_INCORRECT means that the user was
3205 denied access.

3206 If there are any active cryptographic or object finding operations in an application's session, and then
3207 **C_Login** is successfully executed by that application, it may or may not be the case that those operations
3208 are still active. Therefore, before logging in, any active operations should be finished.

3209 If the application calling **C_Login** has a R/O session open with the token, then it will be unable to log the
3210 SO into a session (see [PKCS11-UG] for further details). An attempt to do this will result in the error code
3211 CKR_SESSION_READ_ONLY_EXISTS.

3212 C_Login may be called repeatedly, without intervening **C_Logout** calls, if (and only if) a key with the
3213 CKA_ALWAYS_AUTHENTICATE attribute set to CK_TRUE exists, and the user needs to do
3214 cryptographic operation on this key. See further Section 4.9.

3215 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
 3216 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
 3217 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
 3218 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_PIN_INCORRECT,
 3219 CKR_PIN_LOCKED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
 3220 CKR_SESSION_READ_ONLY_EXISTS, CKR_USER_ALREADY_LOGGED_IN,
 3221 CKR_USER_ANOTHER_ALREADY_LOGGED_IN, CKR_USER_PIN_NOT_INITIALIZED,
 3222 CKR_USER_TOO_MANY_TYPES, CKR_USER_TYPE_INVALID.

3223 Example: see **C_Logout**.

3224 5.6.9 C_LoginUser

```
3225 CK_DECLARE_FUNCTION(CK_RV, C_LoginUser) (
3226     CK_SESSION_HANDLE hSession,
3227     CK_USER_TYPE userType,
3228     CK_UTF8CHAR_PTR pPin,
3229     CK_ULONG ulPinLen,
3230     CK_UTF8CHAR_PTR pUsername,
3231     CK_ULONG ulUsernameLen
3232 );
```

3233 **C_LoginUser** logs a user into a token. *hSession* is a session handle; *userType* is the user type; *pPin*
 3234 points to the user's PIN; *ulPinLen* is the length of the PIN, *pUsername* points to the user name,
 3235 *ulUsernameLen* is the length of the user name. This standard allows PIN and user name values to
 3236 contain any valid UTF8 character, but the token may impose subset restrictions.

3237 When the user type is either CKU_SO or CKU_USER, if the call succeeds, each of the application's
 3238 sessions will enter either the "R/W SO Functions" state, the "R/W User Functions" state, or the "R/O User
 3239 Functions" state. If the user type is CKU_CONTEXT_SPECIFIC, the behavior of **C_LoginUser** depends
 3240 on the context in which it is called. Improper use of this user type will result in a return value
 3241 CKR_OPERATION_NOT_INITIALIZED.

3242 If the token has a "protected authentication path", as indicated by the
 3243 CKF_PROTECTED_AUTHENTICATION_PATH flag in its CK_TOKEN_INFO being set, then that means
 3244 that there is some way for a user to be authenticated to the token without having to send a PIN through
 3245 the Cryptoki library. One such possibility is that the user enters a PIN on a PIN pad on the token itself, or
 3246 on the slot device. The user might not even use a PIN—authentication could be achieved by some
 3247 fingerprint-reading device, for example. To log into a token with a protected authentication path, the *pPin*
 3248 parameter to **C_LoginUser** should be NULL_PTR. When **C_LoginUser** returns, whatever authentication
 3249 method supported by the token will have been performed; a return value of CKR_OK means that the user
 3250 was successfully authenticated, and a return value of CKR_PIN_INCORRECT means that the user was
 3251 denied access.

3252 If there are any active cryptographic or object finding operations in an application's session, and then
 3253 **C_LoginUser** is successfully executed by that application, it may or may not be the case that those
 3254 operations are still active. Therefore, before logging in, any active operations should be finished.

3255 If the application calling **C_LoginUser** has a R/O session open with the token, then it will be unable to log
 3256 the SO into a session (see [PKCS11-UG] for further details). An attempt to do this will result in the error
 3257 code CKR_SESSION_READ_ONLY_EXISTS.

3258 **C_LoginUser** may be called repeatedly, without intervening **C_Logout** calls, if (and only if) a key with the
 3259 CKA_ALWAYS_AUTHENTICATE attribute set to CK_TRUE exists, and the user needs to do
 3260 cryptographic operation on this key. See further Section 4.9.

3261 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
 3262 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
 3263 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
 3264 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_PIN_INCORRECT,
 3265 CKR_PIN_LOCKED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
 3266 CKR_SESSION_READ_ONLY_EXISTS, CKR_USER_ALREADY_LOGGED_IN,

3267 CKR_USER_ANOTHER_ALREADY_LOGGED_IN, CKR_USER_PIN_NOT_INITIALIZED,
3268 CKR_USER_TOO_MANY_TYPES, CKR_USER_TYPE_INVALID.

3269 Example:

```
3270 CK_SESSION_HANDLE hSession;  
3271 CK_UTF8CHAR userPIN[] = {"MyPIN"};  
3272 CK_UTF8CHAR userName[] = {"MyUserName"};  
3273 CK_RV rv;  
3274  
3275 rv = C_LoginUser(hSession, CKU_USER, userPIN, sizeof(userPIN)-1, username,  
3276 sizeof(username)-1);  
3277 if (rv == CKR_OK) {  
3278     .  
3279     .  
3280     rv == C_Logout(hSession);  
3281     if (rv == CKR_OK) {  
3282         .  
3283         .  
3284     }  
3285 }
```

3286 5.6.10 C_Logout

```
3287 CK_DECLARE_FUNCTION(CK_RV, C_Logout) (  
3288     CK_SESSION_HANDLE hSession  
3289 );
```

3290 **C_Logout** logs a user out from a token. *hSession* is the session's handle.

3291 Depending on the current user type, if the call succeeds, each of the application's sessions will enter
3292 either the "R/W Public Session" state or the "R/O Public Session" state.

3293 When **C_Logout** successfully executes, any of the application's handles to private objects become invalid
3294 (even if a user is later logged back into the token, those handles remain invalid). In addition, all private
3295 session objects from sessions belonging to the application are destroyed.

3296 If there are any active cryptographic or object-finding operations in an application's session, and then
3297 **C_Logout** is successfully executed by that application, it may or may not be the case that those
3298 operations are still active. Therefore, before logging out, any active operations should be finished.

3299 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
3300 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
3301 CKR_HOST_MEMORY, CKR_OK, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
3302 CKR_USER_NOT_LOGGED_IN.

3303 Example:

```
3304 CK_SESSION_HANDLE hSession;  
3305 CK_UTF8CHAR userPIN[] = {"MyPIN"};  
3306 CK_RV rv;  
3307  
3308 rv = C_Login(hSession, CKU_USER, userPIN, sizeof(userPIN)-1);  
3309 if (rv == CKR_OK) {  
3310     .
```

```

3311 .
3312 rv == C_Logout(hSession);
3313 if (rv == CKR_OK) {
3314     .
3315     .
3316 }
3317 }

```

3318 5.7 Object management functions

3319 Cryptoki provides the following functions for managing objects. Additional functions provided specifically
3320 for managing key objects are described in Section 5.18.

3321 5.7.1 C_CreateObject

```

3322 CK_DECLARE_FUNCTION(CK_RV, C_CreateObject) (
3323     CK_SESSION_HANDLE hSession,
3324     CK_ATTRIBUTE_PTR pTemplate,
3325     CK_ULONG ulCount,
3326     CK_OBJECT_HANDLE_PTR phObject
3327 );

```

3328 **C_CreateObject** creates a new object. *hSession* is the session's handle; *pTemplate* points to the object's
3329 template; *ulCount* is the number of attributes in the template; *phObject* points to the location that receives
3330 the new object's handle.

3331 If a call to **C_CreateObject** cannot support the precise template supplied to it, it will fail and return without
3332 creating any object.

3333 If **C_CreateObject** is used to create a key object, the key object will have its **CKA_LOCAL** attribute set to
3334 CK_FALSE. If that key object is a secret or private key then the new key will have the
3335 **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, and the **CKA_NEVER_EXTRACTABLE**
3336 attribute set to CK_FALSE.

3337 Only session objects can be created during a read-only session. Only public objects can be created
3338 unless the normal user is logged in.

3339 Whenever an object is created, a value for CKA_UNIQUE_ID is generated and assigned to the new
3340 object (See Section 4.4.1).

3341 Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY,
3342 CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID,
3343 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_CURVE_NOT_SUPPORTED, CKR_DEVICE_ERROR,
3344 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DOMAIN_PARAMS_INVALID,
3345 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
3346 CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
3347 CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCOMPLETE, CKR_TEMPLATE_INCONSISTENT,
3348 CKR_TOKEN_WRITE_PROTECTED, CKR_USER_NOT_LOGGED_IN.

3349 Example:

```

3350 CK_SESSION_HANDLE hSession;
3351 CK_OBJECT_HANDLE
3352     hData,
3353     hCertificate,
3354     hKey;
3355 CK_OBJECT_CLASS
3356     dataClass = CKO_DATA,

```

```

3357     certificateClass = CKO_CERTIFICATE,
3358     keyClass = CKO_PUBLIC_KEY;
3359 CK_KEY_TYPE keyType = CKK_RSA;
3360 CK_UTF8CHAR application[] = {"My Application"};
3361 CK_BYTE dataValue[] = {...};
3362 CK_BYTE subject[] = {...};
3363 CK_BYTE id[] = {...};
3364 CK_BYTE certificateValue[] = {...};
3365 CK_BYTE modulus[] = {...};
3366 CK_BYTE exponent[] = {...};
3367 CK_BBOOL true = CK_TRUE;
3368 CK_ATTRIBUTE dataTemplate[] = {
3369     {CKA_CLASS, &dataClass, sizeof(dataClass)},
3370     {CKA_TOKEN, &true, sizeof(true)},
3371     {CKA_APPLICATION, application, sizeof(application)-1},
3372     {CKA_VALUE, dataValue, sizeof(dataValue)}
3373 };
3374 CK_ATTRIBUTE certificateTemplate[] = {
3375     {CKA_CLASS, &certificateClass, sizeof(certificateClass)},
3376     {CKA_TOKEN, &true, sizeof(true)},
3377     {CKA_SUBJECT, subject, sizeof(subject)},
3378     {CKA_ID, id, sizeof(id)},
3379     {CKA_VALUE, certificateValue, sizeof(certificateValue)}
3380 };
3381 CK_ATTRIBUTE keyTemplate[] = {
3382     {CKA_CLASS, &keyClass, sizeof(keyClass)},
3383     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
3384     {CKA_WRAP, &true, sizeof(true)},
3385     {CKA_MODULUS, modulus, sizeof(modulus)},
3386     {CKA_PUBLIC_EXPONENT, exponent, sizeof(exponent)}
3387 };
3388 CK_RV rv;
3389
3390 .
3391 .
3392 /* Create a data object */
3393 rv = C_CreateObject(hSession, &dataTemplate, 4, &hData);
3394 if (rv == CKR_OK) {
3395     .
3396     .
3397 }
3398
3399 /* Create a certificate object */

```

```

3400 rv = C_CreateObject(
3401     hSession, &certificateTemplate, 5, &hCertificate);
3402 if (rv == CKR_OK) {
3403     .
3404     .
3405 }
3406
3407 /* Create an RSA public key object */
3408 rv = C_CreateObject(hSession, &keyTemplate, 5, &hKey);
3409 if (rv == CKR_OK) {
3410     .
3411     .
3412 }

```

3413 5.7.2 C_CopyObject

```

3414 CK_DECLARE_FUNCTION(CK_RV, C_CopyObject)(
3415     CK_SESSION_HANDLE hSession,
3416     CK_OBJECT_HANDLE hObject,
3417     CK_ATTRIBUTE_PTR pTemplate,
3418     CK_ULONG ulCount,
3419     CK_OBJECT_HANDLE_PTR phNewObject
3420 );

```

3421 **C_CopyObject** copies an object, creating a new object for the copy. *hSession* is the session's handle;
3422 *hObject* is the object's handle; *pTemplate* points to the template for the new object; *ulCount* is the number
3423 of attributes in the template; *phNewObject* points to the location that receives the handle for the copy of
3424 the object.

3425 The template may specify new values for any attributes of the object that can ordinarily be modified (e.g.,
3426 in the course of copying a secret key, a key's **CKA_EXTRACTABLE** attribute may be changed from
3427 CK_TRUE to CK_FALSE, but not the other way around. If this change is made, the new key's
3428 **CKA_NEVER_EXTRACTABLE** attribute will have the value CK_FALSE. Similarly, the template may
3429 specify that the new key's **CKA_SENSITIVE** attribute be CK_TRUE; the new key will have the same
3430 value for its **CKA_ALWAYS_SENSITIVE** attribute as the original key). It may also specify new values of
3431 the **CKA_TOKEN** and **CKA_PRIVATE** attributes (e.g., to copy a session object to a token object). If the
3432 template specifies a value of an attribute which is incompatible with other existing attributes of the object,
3433 the call fails with the return code CKR_TEMPLATE_INCONSISTENT.

3434 If a call to **C_CopyObject** cannot support the precise template supplied to it, it will fail and return without
3435 creating any object. If the object indicated by *hObject* has its CKA_COPYABLE attribute set to
3436 CK_FALSE, C_CopyObject will return CKR_ACTION_PROHIBITED.

3437 Whenever an object is copied, a new value for CKA_UNIQUE_ID is generated and assigned to the new
3438 object (See Section 4.4.1).

3439 Only session objects can be created during a read-only session. Only public objects can be created
3440 unless the normal user is logged in.

3441 Return values: , CKR_ACTION_PROHIBITED, CKR_ARGUMENTS_BAD,
3442 CKR_ATTRIBUTE_READ_ONLY, CKR_ATTRIBUTE_TYPE_INVALID,
3443 CKR_ATTRIBUTE_VALUE_INVALID, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR,
3444 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED,
3445 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OBJECT_HANDLE_INVALID, CKR_OK,
3446 CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
3447 CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCONSISTENT,
3448 CKR_TOKEN_WRITE_PROTECTED, CKR_USER_NOT_LOGGED_IN.

3449 Example:

```
3450 CK_SESSION_HANDLE hSession;
3451 CK_OBJECT_HANDLE hKey, hNewKey;
3452 CK_OBJECT_CLASS keyClass = CKO_SECRET_KEY;
3453 CK_KEY_TYPE keyType = CKK_DES;
3454 CK_BYTE id[] = {...};
3455 CK_BYTE keyValue[] = {...};
3456 CK_BBOOL false = CK_FALSE;
3457 CK_BBOOL true = CK_TRUE;
3458 CK_ATTRIBUTE keyTemplate[] = {
3459     {CKA_CLASS, &keyClass, sizeof(keyClass)},
3460     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
3461     {CKA_TOKEN, &false, sizeof(false)},
3462     {CKA_ID, id, sizeof(id)},
3463     {CKA_VALUE, keyValue, sizeof(keyValue)}
3464 };
3465 CK_ATTRIBUTE copyTemplate[] = {
3466     {CKA_TOKEN, &true, sizeof(true)}
3467 };
3468 CK_RV rv;
3469
3470 .
3471 .
3472 /* Create a DES secret key session object */
3473 rv = C_CreateObject(hSession, &keyTemplate, 5, &hKey);
3474 if (rv == CKR_OK) {
3475     /* Create a copy which is a token object */
3476     rv = C_CopyObject(hSession, hKey, &copyTemplate, 1, &hNewKey);
3477     .
3478     .
3479 }
```

3480 5.7.3 C_DestroyObject

```
3481 CK_DECLARE_FUNCTION(CK_RV, C_DestroyObject) (
3482     CK_SESSION_HANDLE hSession,
3483     CK_OBJECT_HANDLE hObject
3484 );
```

3485 **C_DestroyObject** destroys an object. *hSession* is the session's handle; and *hObject* is the object's handle.

3487 Only session objects can be destroyed during a read-only session. Only public objects can be destroyed unless the normal user is logged in.

3489 Certain objects may not be destroyed. Calling **C_DestroyObject** on such objects will result in the
3490 CKR_ACTION_PROHIBITED error code. An application can consult the object's CKA_DESTROYABLE
3491 attribute to determine if an object may be destroyed or not.

3492 Return values: , CKR_ACTION_PROHIBITED, CKR_CRYPTOKI_NOT_INITIALIZED,
 3493 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
 3494 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY,
 3495 CKR_OBJECT_HANDLE_INVALID, CKR_OK, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED,
 3496 CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY,
 3497 CKR_TOKEN_WRITE_PROTECTED.

3498 Example: see **C_GetObjectSize**.

3499 5.7.4 C_GetObjectSize

```
3500 CK_DECLARE_FUNCTION(CK_RV, C_GetObjectSize)(
3501     CK_SESSION_HANDLE hSession,
3502     CK_OBJECT_HANDLE hObject,
3503     CK_ULONG_PTR pulSize
3504 );
```

3505 **C_GetObjectSize** gets the size of an object in bytes. *hSession* is the session's handle; *hObject* is the
 3506 object's handle; *pulSize* points to the location that receives the size in bytes of the object.

3507 Cryptoki does not specify what the precise meaning of an object's size is. Intuitively, it is some measure
 3508 of how much token memory the object takes up. If an application deletes (say) a private object of size *S*,
 3509 it might be reasonable to assume that the *ulFreePrivateMemory* field of the token's **CK_TOKEN_INFO**
 3510 structure increases by approximately *S*.

3511 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
 3512 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
 3513 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY,
 3514 CKR_INFORMATION_SENSITIVE, CKR_OBJECT_HANDLE_INVALID, CKR_OK,
 3515 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

3516 Example:

```
3517 CK_SESSION_HANDLE hSession;
3518 CK_OBJECT_HANDLE hObject;
3519 CK_OBJECT_CLASS dataClass = CKO_DATA;
3520 CK_UTF8CHAR application[] = {"My Application"};
3521 CK_BYTE dataValue[] = {...};
3522 CK_BYTE value[] = {...};
3523 CK_BBOOL true = CK_TRUE;
3524 CK_ATTRIBUTE template[] = {
3525     {CKA_CLASS, &dataClass, sizeof(dataClass)},
3526     {CKA_TOKEN, &true, sizeof(true)},
3527     {CKA_APPLICATION, application, sizeof(application)-1},
3528     {CKA_VALUE, value, sizeof(value)}
3529 };
3530 CK_ULONG ulSize;
3531 CK_RV rv;
3532
3533 .
3534 .
3535 rv = C_CreateObject(hSession, &template, 4, &hObject);
3536 if (rv == CKR_OK) {
3537     rv = C_GetObjectSize(hSession, hObject, &ulSize);
```



```

3538     if (rv != CKR_INFORMATION_SENSITIVE) {
3539         .
3540         .
3541     }
3542
3543     rv = C_DestroyObject(hSession, hObject);
3544     .
3545     .
3546 }

```

3547 5.7.5 C_GetAttributeValue

```

3548 CK_DECLARE_FUNCTION(CK_RV, C_GetAttributeValue) (
3549     CK_SESSION_HANDLE hSession,
3550     CK_OBJECT_HANDLE hObject,
3551     CK_ATTRIBUTE_PTR pTemplate,
3552     CK_ULONG ulCount
3553 );

```

3554 **C_GetAttributeValue** obtains the value of one or more attributes of an object. *hSession* is the session's
3555 handle; *hObject* is the object's handle; *pTemplate* points to a template that specifies which attribute
3556 values are to be obtained, and receives the attribute values; *ulCount* is the number of attributes in the
3557 template.

3558 For each (*type*, *pValue*, *ulValueLen*) triple in the template, **C_GetAttributeValue** performs the following
3559 algorithm:

- 3560 1. If the specified attribute (i.e., the attribute specified by the *type* field) for the object cannot be revealed
3561 because the object is sensitive or unextractable, then the *ulValueLen* field in that triple is modified to
3562 hold the value CK_UNAVAILABLE_INFORMATION.
- 3563 2. Otherwise, if the specified value for the object is invalid (the object does not possess such an
3564 attribute), then the *ulValueLen* field in that triple is modified to hold the value
3565 CK_UNAVAILABLE_INFORMATION.
- 3566 3. Otherwise, if the *pValue* field has the value NULL_PTR, then the *ulValueLen* field is modified to hold
3567 the exact length of the specified attribute for the object.
- 3568 4. Otherwise, if the length specified in *ulValueLen* is large enough to hold the value of the specified
3569 attribute for the object, then that attribute is copied into the buffer located at *pValue*, and the
3570 *ulValueLen* field is modified to hold the exact length of the attribute.
- 3571 5. Otherwise, the *ulValueLen* field is modified to hold the value CK_UNAVAILABLE_INFORMATION.

3572 If case 1 applies to any of the requested attributes, then the call should return the value
3573 CKR_ATTRIBUTE_SENSITIVE. If case 2 applies to any of the requested attributes, then the call should
3574 return the value CKR_ATTRIBUTE_TYPE_INVALID. If case 5 applies to any of the requested attributes,
3575 then the call should return the value CKR_BUFFER_TOO_SMALL. As usual, if more than one of these
3576 error codes is applicable, Cryptoki may return any of them. Only if none of them applies to any of the
3577 requested attributes will CKR_OK be returned.

3578 In the special case of an attribute whose value is an array of attributes, for example
3579 CKA_WRAP_TEMPLATE, where it is passed in with *pValue* not NULL, the length specified in *ulValueLen*
3580 MUST be large enough to hold all attributes in the array. If the *pValue* of elements within the array is
3581 NULL_PTR then the *ulValueLen* of elements within the array will be set to the required length. If the
3582 *pValue* of elements within the array is not NULL_PTR, then the *ulValueLen* element of attributes within
3583 the array MUST reflect the space that the corresponding *pValue* points to, and *pValue* is filled in if there is
3584 sufficient room. Therefore it is important to initialize the contents of a buffer before calling
3585 C_GetAttributeValue to get such an array value. Note that the *type* element of attributes within the array
3586 MUST be ignored on input and MUST be set on output. If any *ulValueLen* within the array isn't large

3587 enough, it will be set to CK_UNAVAILABLE_INFORMATION and the function will return
3588 CKR_BUFFER_TOO_SMALL, as it does if an attribute in the pTemplate argument has ulValueLen too
3589 small. Note that any attribute whose value is an array of attributes is identifiable by virtue of the attribute
3590 type having the CKF_ARRAY_ATTRIBUTE bit set.

3591 Note that the error codes CKR_ATTRIBUTE_SENSITIVE, CKR_ATTRIBUTE_TYPE_INVALID, and
3592 CKR_BUFFER_TOO_SMALL do not denote true errors for **C_GetAttributeValue**. If a call to
3593 **C_GetAttributeValue** returns any of these three values, then the call MUST nonetheless have processed
3594 every attribute in the template supplied to **C_GetAttributeValue**. Each attribute in the template whose
3595 value *can be* returned by the call to **C_GetAttributeValue** *will be* returned by the call to
3596 **C_GetAttributeValue**.

3597 Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_SENSITIVE,
3598 CKR_ATTRIBUTE_TYPE_INVALID, CKR_BUFFER_TOO_SMALL,
3599 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
3600 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
3601 CKR_HOST_MEMORY, CKR_OBJECT_HANDLE_INVALID, CKR_OK, CKR_SESSION_CLOSED,
3602 CKR_SESSION_HANDLE_INVALID.

3603 Example:

```
3604 CK_SESSION_HANDLE hSession;  
3605 CK_OBJECT_HANDLE hObject;  
3606 CK_BYTE_PTR pModulus, pExponent;  
3607 CK_ATTRIBUTE template[] = {  
3608     {CKA_MODULUS, NULL_PTR, 0},  
3609     {CKA_PUBLIC_EXPONENT, NULL_PTR, 0}  
3610 };  
3611 CK_RV rv;  
3612  
3613 .  
3614 .  
3615 rv = C_GetAttributeValue(hSession, hObject, &template, 2);  
3616 if (rv == CKR_OK) {  
3617     pModulus = (CK_BYTE_PTR) malloc(template[0].ulValueLen);  
3618     template[0].pValue = pModulus;  
3619     /* template[0].ulValueLen was set by C_GetAttributeValue */  
3620  
3621     pExponent = (CK_BYTE_PTR) malloc(template[1].ulValueLen);  
3622     template[1].pValue = pExponent;  
3623     /* template[1].ulValueLen was set by C_GetAttributeValue */  
3624  
3625     rv = C_GetAttributeValue(hSession, hObject, &template, 2);  
3626     if (rv == CKR_OK) {  
3627         .  
3628         .  
3629     }  
3630     free(pModulus);  
3631     free(pExponent);  
3632 }
```

5.7.6 C_SetAttributeValue

```
CK_DECLARE_FUNCTION(CK_RV, C_SetAttributeValue)(
    CK_SESSION_HANDLE hSession,
    CK_OBJECT_HANDLE hObject,
    CK_ATTRIBUTE_PTR pTemplate,
    CK_ULONG ulCount
);
```

C_SetAttributeValue modifies the value of one or more attributes of an object. *hSession* is the session's handle; *hObject* is the object's handle; *pTemplate* points to a template that specifies which attribute values are to be modified and their new values; *ulCount* is the number of attributes in the template.

Certain objects may not be modified. Calling **C_SetAttributeValue** on such objects will result in the **CKR_ACTION_PROHIBITED** error code. An application can consult the object's **CKA_MODIFIABLE** attribute to determine if an object may be modified or not.

Only session objects can be modified during a read-only session.

The template may specify new values for any attributes of the object that can be modified. If the template specifies a value of an attribute which is incompatible with other existing attributes of the object, the call fails with the return code **CKR_TEMPLATE_INCONSISTENT**.

Not all attributes can be modified; see Section 4.1.2 for more details.

Return values: **CKR_ACTION_PROHIBITED**, **CKR_ARGUMENTS_BAD**, **CKR_ATTRIBUTE_READ_ONLY**, **CKR_ATTRIBUTE_TYPE_INVALID**, **CKR_ATTRIBUTE_VALUE_INVALID**, **CKR_CRYPTOKI_NOT_INITIALIZED**, **CKR_DEVICE_ERROR**, **CKR_DEVICE_MEMORY**, **CKR_DEVICE_REMOVED**, **CKR_FUNCTION_FAILED**, **CKR_GENERAL_ERROR**, **CKR_HOST_MEMORY**, **CKR_OBJECT_HANDLE_INVALID**, **CKR_OK**, **CKR_SESSION_CLOSED**, **CKR_SESSION_HANDLE_INVALID**, **CKR_SESSION_READ_ONLY**, **CKR_TEMPLATE_INCONSISTENT**, **CKR_TOKEN_WRITE_PROTECTED**, **CKR_USER_NOT_LOGGED_IN**.

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hObject;
CK_UTF8CHAR label[] = {"New label"};
CK_ATTRIBUTE template[] = {
    CKA_LABEL, label, sizeof(label)-1
};
CK_RV rv;

.
.

rv = C_SetAttributeValue(hSession, hObject, &template, 1);
if (rv == CKR_OK) {
    .
    .
}
```

5.7.7 C_FindObjectsInit

```
CK_DECLARE_FUNCTION(CK_RV, C_FindObjectsInit)(
    CK_SESSION_HANDLE hSession,
    CK_ATTRIBUTE_PTR pTemplate,
```

```
3679     CK_ULONG ulCount
3680 );
```

3681 **C_FindObjectsInit** initializes a search for token and session objects that match a template. *hSession* is
3682 the session's handle; *pTemplate* points to a search template that specifies the attribute values to match;
3683 *ulCount* is the number of attributes in the search template. The matching criterion is an exact byte-for-
3684 byte match with all attributes in the template. To find all objects, set *ulCount* to 0.

3685 After calling **C_FindObjectsInit**, the application may call **C_FindObjects** one or more times to obtain
3686 handles for objects matching the template, and then eventually call **C_FindObjectsFinal** to finish the
3687 active search operation. At most one search operation may be active at a given time in a given session.

3688 The object search operation will only find objects that the session can view. For example, an object
3689 search in an "R/W Public Session" will not find any private objects (even if one of the attributes in the
3690 search template specifies that the search is for private objects).

3691 If a search operation is active, and objects are created or destroyed which fit the search template for the
3692 active search operation, then those objects may or may not be found by the search operation. Note that
3693 this means that, under these circumstances, the search operation may return invalid object handles.

3694 Even though **C_FindObjectsInit** can return the values CKR_ATTRIBUTE_TYPE_INVALID and
3695 CKR_ATTRIBUTE_VALUE_INVALID, it is not required to. For example, if it is given a search template
3696 with nonexistent attributes in it, it can return CKR_ATTRIBUTE_TYPE_INVALID, or it can initialize a
3697 search operation which will match no objects and return CKR_OK.

3698 If the CKA_UNIQUE_ID attribute is present in the search template, either zero or one objects will be
3699 found, since at most one object can have any particular CKA_UNIQUE_ID value.

3700 Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_TYPE_INVALID,
3701 CKR_ATTRIBUTE_VALUE_INVALID, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR,
3702 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED,
3703 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE,
3704 CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

3705 Example: see **C_FindObjectsFinal**.

3706 5.7.8 C_FindObjects

```
3707 CK_DECLARE_FUNCTION(CK_RV, C_FindObjects)(
3708     CK_SESSION_HANDLE hSession,
3709     CK_OBJECT_HANDLE_PTR phObject,
3710     CK_ULONG ulMaxObjectCount,
3711     CK_ULONG_PTR pulObjectCount
3712 );
```

3713 **C_FindObjects** continues a search for token and session objects that match a template, obtaining
3714 additional object handles. *hSession* is the session's handle; *phObject* points to the location that receives
3715 the list (array) of additional object handles; *ulMaxObjectCount* is the maximum number of object handles
3716 to be returned; *pulObjectCount* points to the location that receives the actual number of object handles
3717 returned.

3718 If there are no more objects matching the template, then the location that *pulObjectCount* points to
3719 receives the value 0.

3720 The search MUST have been initialized with **C_FindObjectsInit**.

3721 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
3722 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
3723 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
3724 CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

3725 Example: see **C_FindObjectsFinal**.

5.7.9 C_FindObjectsFinal

```
CK_DECLARE_FUNCTION(CK_RV, C_FindObjectsFinal)(
    CK_SESSION_HANDLE hSession
);
```

C_FindObjectsFinal terminates a search for token and session objects. *hSession* is the session's handle.

Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hObject;
CK_ULONG ulObjectCount;
CK_RV rv;

.
.
rv = C_FindObjectsInit(hSession, NULL_PTR, 0);
assert(rv == CKR_OK);
while (1) {
    rv = C_FindObjects(hSession, &hObject, 1, &ulObjectCount);
    if (rv != CKR_OK || ulObjectCount == 0)
        break;
    .
    .
}

rv = C_FindObjectsFinal(hSession);
assert(rv == CKR_OK);
```

5.8 Encryption functions

Cryptoki provides the following functions for encrypting data:

5.8.1 C_EncryptInit

```
CK_DECLARE_FUNCTION(CK_RV, C_EncryptInit)(
    CK_SESSION_HANDLE hSession,
    CK_MECHANISM_PTR pMechanism,
    CK_OBJECT_HANDLE hKey
);
```

C_EncryptInit initializes an encryption operation. *hSession* is the session's handle; *pMechanism* points to the encryption mechanism; *hKey* is the handle of the encryption key.

The **CKA_ENCRYPT** attribute of the encryption key, which indicates whether the key supports encryption, MUST be CK_TRUE.

3768 After calling **C_EncryptInit**, the application can either call **C_Encrypt** to encrypt data in a single part; or
3769 call **C_EncryptUpdate** zero or more times, followed by **C_EncryptFinal**, to encrypt data in multiple parts.
3770 The encryption operation is active until the application uses a call to **C_Encrypt** or **C_EncryptFinal** to
3771 *actually obtain* the final piece of ciphertext. To process additional data (in single or multiple parts), the
3772 application MUST call **C_EncryptInit** again.

3773 **C_EncryptInit** can be called with *pMechanism* set to NULL_PTR to terminate an active encryption
3774 operation. If an active operation operations cannot be cancelled, CKR_OPERATION_CANCEL_FAILED
3775 must be returned.

3776 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
3777 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
3778 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED,
3779 CKR_KEY_HANDLE_INVALID, CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT,
3780 CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK,
3781 CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED,
3782 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
3783 CKR_OPERATION_CANCEL_FAILED.

3784 Example: see **C_EncryptFinal**.

3785 5.8.2 C_Encrypt

```
3786 CK_DECLARE_FUNCTION(CK_RV, C_Encrypt) (  
3787     CK_SESSION_HANDLE hSession,  
3788     CK_BYTE_PTR pData,  
3789     CK_ULONG ulDataLen,  
3790     CK_BYTE_PTR pEncryptedData,  
3791     CK_ULONG_PTR pulEncryptedDataLen  
3792 );
```

3793 **C_Encrypt** encrypts single-part data. *hSession* is the session's handle; *pData* points to the data;
3794 *ulDataLen* is the length in bytes of the data; *pEncryptedData* points to the location that receives the
3795 encrypted data; *pulEncryptedDataLen* points to the location that holds the length in bytes of the encrypted
3796 data.

3797 **C_Encrypt** uses the convention described in Section 5.2 on producing output.

3798 The encryption operation MUST have been initialized with **C_EncryptInit**. A call to **C_Encrypt** always
3799 terminates the active encryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a
3800 successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the
3801 ciphertext.

3802 **C_Encrypt** cannot be used to terminate a multi-part operation, and MUST be called after **C_EncryptInit**
3803 without intervening **C_EncryptUpdate** calls.

3804 For some encryption mechanisms, the input plaintext data has certain length constraints (either because
3805 the mechanism can only encrypt relatively short pieces of plaintext, or because the mechanism's input
3806 data MUST consist of an integral number of blocks). If these constraints are not satisfied, then
3807 **C_Encrypt** will fail with return code CKR_DATA_LEN_RANGE.

3808 The plaintext and ciphertext can be in the same place, *i.e.*, it is OK if *pData* and *pEncryptedData* point to
3809 the same location.

3810 For most mechanisms, **C_Encrypt** is equivalent to a sequence of **C_EncryptUpdate** operations followed
3811 by **C_EncryptFinal**.

3812 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
3813 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE,
3814 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
3815 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
3816 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
3817 CKR_SESSION_HANDLE_INVALID.

3818 Example: see **C_EncryptFinal** for an example of similar functions.

5.8.3 C_EncryptUpdate

```
CK_DECLARE_FUNCTION(CK_RV, C_EncryptUpdate) (
    CK_SESSION_HANDLE hSession,
    CK_BYTE_PTR pPart,
    CK_ULONG ulPartLen,
    CK_BYTE_PTR pEncryptedPart,
    CK_ULONG_PTR pulEncryptedPartLen
);
```

C_EncryptUpdate continues a multiple-part encryption operation, processing another data part. *hSession* is the session's handle; *pPart* points to the data part; *ulPartLen* is the length of the data part; *pEncryptedPart* points to the location that receives the encrypted data part; *pulEncryptedPartLen* points to the location that holds the length in bytes of the encrypted data part.

C_EncryptUpdate uses the convention described in Section 5.2 on producing output.

The encryption operation MUST have been initialized with **C_EncryptInit**. This function may be called any number of times in succession. A call to **C_EncryptUpdate** which results in an error other than CKR_BUFFER_TOO_SMALL terminates the current encryption operation.

The plaintext and ciphertext can be in the same place, i.e., it is OK if *pPart* and *pEncryptedPart* point to the same location.

Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example: see **C_EncryptFinal**.

5.8.4 C_EncryptFinal

```
CK_DECLARE_FUNCTION(CK_RV, C_EncryptFinal) (
    CK_SESSION_HANDLE hSession,
    CK_BYTE_PTR pLastEncryptedPart,
    CK_ULONG_PTR pulLastEncryptedPartLen
);
```

C_EncryptFinal finishes a multiple-part encryption operation. *hSession* is the session's handle; *pLastEncryptedPart* points to the location that receives the last encrypted data part, if any; *pulLastEncryptedPartLen* points to the location that holds the length of the last encrypted data part.

C_EncryptFinal uses the convention described in Section 5.2 on producing output.

The encryption operation MUST have been initialized with **C_EncryptInit**. A call to **C_EncryptFinal** always terminates the active encryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful call (i.e., one which returns CKR_OK) to determine the length of the buffer needed to hold the ciphertext.

For some multi-part encryption mechanisms, the input plaintext data has certain length constraints, because the mechanism's input data MUST consist of an integral number of blocks. If these constraints are not satisfied, then **C_EncryptFinal** will fail with return code CKR_DATA_LEN_RANGE.

Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example:

```
#define PLAINTEXT_BUF_SZ 200
#define CIPHERTEXT_BUF_SZ 256
```



```

3868
3869 CK_ULONG firstPieceLen, secondPieceLen;
3870 CK_SESSION_HANDLE hSession;
3871 CK_OBJECT_HANDLE hKey;
3872 CK_BYTE iv[8];
3873 CK_MECHANISM mechanism = {
3874     CKM_DES_CBC_PAD, iv, sizeof(iv)
3875 };
3876 CK_BYTE data[PLAINTEXT_BUF_SZ];
3877 CK_BYTE encryptedData[CIPHERTEXT_BUF_SZ];
3878 CK_ULONG ulEncryptedData1Len;
3879 CK_ULONG ulEncryptedData2Len;
3880 CK_ULONG ulEncryptedData3Len;
3881 CK_RV rv;
3882
3883 .
3884 .
3885 firstPieceLen = 90;
3886 secondPieceLen = PLAINTEXT_BUF_SZ-firstPieceLen;
3887 rv = C_EncryptInit(hSession, &mechanism, hKey);
3888 if (rv == CKR_OK) {
3889     /* Encrypt first piece */
3890     ulEncryptedData1Len = sizeof(encryptedData);
3891     rv = C_EncryptUpdate(
3892         hSession,
3893         &data[0], firstPieceLen,
3894         &encryptedData[0], &ulEncryptedData1Len);
3895     if (rv != CKR_OK) {
3896         .
3897         .
3898     }
3899
3900     /* Encrypt second piece */
3901     ulEncryptedData2Len = sizeof(encryptedData)-ulEncryptedData1Len;
3902     rv = C_EncryptUpdate(
3903         hSession,
3904         &data[firstPieceLen], secondPieceLen,
3905         &encryptedData[ulEncryptedData1Len], &ulEncryptedData2Len);
3906     if (rv != CKR_OK) {
3907         .
3908         .
3909     }
3910

```



```

3911  /* Get last little encrypted bit */
3912  ulEncryptedData3Len =
3913      sizeof(encryptedData)-ulEncryptedData1Len-ulEncryptedData2Len;
3914  rv = C_EncryptFinal(
3915      hSession,
3916      &encryptedData[ulEncryptedData1Len+ulEncryptedData2Len],
3917      &ulEncryptedData3Len);
3918  if (rv != CKR_OK) {
3919      .
3920      .
3921  }
3922  }

```

5.9 Message-based encryption functions

Message-based encryption refers to the process of encrypting multiple messages using the same encryption mechanism and encryption key. The encryption mechanism can be either an authenticated encryption with associated data (AEAD) algorithm or a pure encryption algorithm.

Cryptoki provides the following functions for message-based encryption:

5.9.1 C_MessageEncryptInit

```

CK_DECLARE_FUNCTION(CK_RV, C_MessageEncryptInit)(
    CK_SESSION_HANDLE hSession,
    CK_MECHANISM_PTR pMechanism,
    CK_OBJECT_HANDLE hKey
);

```

C_MessageEncryptInit prepares a session for one or more encryption operations that use the same encryption mechanism and encryption key. *hSession* is the session's handle; *pMechanism* points to the encryption mechanism; *hKey* is the handle of the encryption key.

The CKA_ENCRYPT attribute of the encryption key, which indicates whether the key supports encryption, MUST be CK_TRUE.

After calling **C_MessageEncryptInit**, the application can either call **C_EncryptMessage** to encrypt a message in a single part, or call **C_EncryptMessageBegin**, followed by **C_EncryptMessageNext** one or more times, to encrypt a message in multiple parts. This may be repeated several times. The message-based encryption process is active until the application calls **C_MessageEncryptFinal** to finish the message-based encryption process.

C_MessageEncryptInit can be called with *pMechanism* set to NULL_PTR to terminate a message-based encryption process. If a multi-part message encryption operation is active, it will also be terminated. If an active operation has been initialized and it cannot be cancelled, CKR_OPERATION_CANCEL_FAILED must be returned.

Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID, CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN, CKR_OPERATION_CANCEL_FAILED.

5.9.2 C_EncryptMessage

```
CK_DECLARE_FUNCTION(CK_RV, C_EncryptMessage) (
    CK_SESSION_HANDLE hSession,
    CK_VOID_PTR pParameter,
    CK_ULONG ulParameterLen,
    CK_BYTE_PTR pAssociatedData,
    CK_ULONG ulAssociatedDataLen,
    CK_BYTE_PTR pPlaintext,
    CK_ULONG ulPlaintextLen,
    CK_BYTE_PTR pCiphertext,
    CK_ULONG_PTR pulCiphertextLen
);
```

C_EncryptMessage encrypts a message in a single part. *hSession* is the session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message encryption operation; *pAssociatedData* and *ulAssociatedDataLen* specify the associated data for an AEAD mechanism; *pPlaintext* points to the plaintext data; *ulPlaintextLen* is the length in bytes of the plaintext data; *pCiphertext* points to the location that receives the encrypted data; *pulCiphertextLen* points to the location that holds the length in bytes of the encrypted data.

Typically, *pParameter* is an initialization vector (IV) or nonce. Depending on the mechanism parameter passed to **C_MessageEncryptInit**, *pParameter* may be either an input or an output parameter. For example, if the mechanism parameter specifies an IV generator mechanism, the IV generated by the IV generator will be output to the *pParameter* buffer.

If the encryption mechanism is not AEAD, *pAssociatedData* and *ulAssociatedDataLen* are not used and should be set to (NULL, 0).

C_EncryptMessage uses the convention described in Section 5.2 on producing output.

The message-based encryption process MUST have been initialized with **C_MessageEncryptInit**. A call to **C_EncryptMessage** begins and terminates a message encryption operation.

C_EncryptMessage cannot be called in the middle of a multi-part message encryption operation.

For some encryption mechanisms, the input plaintext data has certain length constraints (either because the mechanism can only encrypt relatively short pieces of plaintext, or because the mechanism's input data MUST consist of an integral number of blocks). If these constraints are not satisfied, then **C_EncryptMessage** will fail with return code CKR_DATA_LEN_RANGE. The plaintext and ciphertext can be in the same place, i.e., it is OK if *pPlaintext* and *pCiphertext* point to the same location.

For most mechanisms, **C_EncryptMessage** is equivalent to **C_EncryptMessageBegin** followed by a sequence of **C_EncryptMessageNext** operations.

Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

5.9.3 C_EncryptMessageBegin

```
CK_DECLARE_FUNCTION(CK_RV, C_EncryptMessageBegin) (
    CK_SESSION_HANDLE hSession,
    CK_VOID_PTR pParameter,
    CK_ULONG ulParameterLen,
    CK_BYTE_PTR pAssociatedData,
    CK_ULONG ulAssociatedDataLen
);
```

C_EncryptMessageBegin begins a multiple-part message encryption operation. *hSession* is the session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the

4006 message encryption operation; *pAssociatedData* and *ulAssociatedDataLen* specify the associated data
 4007 for an AEAD mechanism.

4008 Typically, *pParameter* is an initialization vector (IV) or nonce. Depending on the mechanism parameter
 4009 passed to **C_MessageEncryptInit**, *pParameter* may be either an input or an output parameter. For
 4010 example, if the mechanism parameter specifies an IV generator mechanism, the IV generated by the IV
 4011 generator will be output to the *pParameter* buffer.

4012 If the mechanism is not AEAD, *pAssociatedData* and *ulAssociatedDataLen* are not used and should be
 4013 set to (NULL, 0).

4014 After calling **C_EncryptMessageBegin**, the application should call **C_EncryptMessageNext** one or
 4015 more times to encrypt the message in multiple parts. The message encryption operation is active until the
 4016 application uses a call to **C_EncryptMessageNext** with flags=CKF_END_OF_MESSAGE to actually
 4017 obtain the final piece of ciphertext. To process additional messages (in single or multiple parts), the
 4018 application MUST call **C_EncryptMessage** or **C_EncryptMessageBegin** again.

4019 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
 4020 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
 4021 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE,
 4022 CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
 4023 CKR_USER_NOT_LOGGED_IN.

4024 5.9.4 C_EncryptMessageNext

```

4025 CK_DECLARE_FUNCTION(CK_RV, C_EncryptMessageNext) (
4026     CK_SESSION_HANDLE hSession,
4027     CK_BYTE_PTR pPlaintextPart,
4028     CK_ULONG ulPlaintextPartLen,
4029     CK_BYTE_PTR pCiphertextPart,
4030     CK_ULONG_PTR pulCiphertextPartLen,
4031     CK_ULONG flags
4032 );
  
```

4033 **C_EncryptMessageNext** continues a multiple-part message encryption operation, processing another
 4034 message part. *hSession* is the session's handle; *pPlaintextPart* points to the plaintext message part;
 4035 *ulPlaintextPartLen* is the length of the plaintext message part; *pCiphertextPart* points to the location that
 4036 receives the encrypted message part; *pulCiphertextPartLen* points to the location that holds the length in
 4037 bytes of the encrypted message part; flags is set to 0 if there is more plaintext data to follow, or set to
 4038 CKF_END_OF_MESSAGE if this is the last plaintext part.

4039 **C_EncryptMessageNext** uses the convention described in Section 5.2 on producing output.

4040 The message encryption operation MUST have been started with **C_EncryptMessageBegin**. This
 4041 function may be called any number of times in succession. A call to **C_EncryptMessageNext** with flags=0
 4042 which results in an error other than CKR_BUFFER_TOO_SMALL terminates the current message
 4043 encryption operation. A call to **C_EncryptMessageNext** with flags=CKF_END_OF_MESSAGE always
 4044 terminates the active message encryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a
 4045 successful call (i.e., one which returns **CKR_OK**) to determine the length of the buffer needed to hold the
 4046 ciphertext.

4047 Although the last **C_EncryptMessageNext** call ends the encryption of a message, it does not finish the
 4048 message-based encryption process. Additional **C_EncryptMessage** or **C_EncryptMessageBegin** and
 4049 **C_EncryptMessageNext** calls may be made on the session.

4050 The plaintext and ciphertext can be in the same place, i.e., it is OK if *pPlaintextPart* and *pCiphertextPart*
 4051 point to the same location.

4052 For some multi-part encryption mechanisms, the input plaintext data has certain length constraints,
 4053 because the mechanism's input data MUST consist of an integral number of blocks. If these constraints
 4054 are not satisfied when the final message part is supplied (i.e., with flags=CKF_END_OF_MESSAGE),
 4055 then **C_EncryptMessageNext** will fail with return code CKR_DATA_LEN_RANGE.

4056 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4057 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR,
4058 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED,
4059 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
4060 CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

4061 5.9.5 C_EncryptMessageFinal

```
4062 CK_DECLARE_FUNCTION(CK_RV, C_EncryptMessageNext) (  
4063     CK_SESSION_HANDLE hSession  
4064 );
```

4065 **C_MessageEncryptFinal** finishes a message-based encryption process. hSession is the session's
4066 handle.

4067 The message-based encryption process MUST have been initialized with **C_MessageEncryptInit**.

4068 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4069 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4070 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4071 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
4072 CKR_SESSION_HANDLE_INVALID.

4073 Example:

```
4074 #define PLAINTEXT_BUF_SZ 200  
4075 #define AUTH_BUF_SZ 100  
4076 #define CIPHERTEXT_BUF_SZ 256  
4077  
4078 CK_SESSION_HANDLE hSession;  
4079 CK_OBJECT_HANDLE hKey;  
4080 CK_BYTE iv[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };  
4081 CK_BYTE tag[16];  
4082 CK_GCM_MESSAGE_PARAMS gcmParams = {  
4083     &iv,  
4084     sizeof(iv) * 8,  
4085     0,  
4086     CKG_NO_GENERATE,  
4087     &tag,  
4088     sizeof(tag) * 8  
4089 };  
4090 CK_MECHANISM mechanism = {  
4091     CKM_AES_GCM, &gcmParams, sizeof(gcmParams)  
4092 };  
4093 CK_BYTE data[2][PLAINTEXT_BUF_SZ];  
4094 CK_BYTE auth[2][AUTH_BUF_SZ];  
4095 CK_BYTE encryptedData[2][CIPHERTEXT_BUF_SZ];  
4096 CK_ULONG ulEncryptedDataLen, ulFirstEncryptedDataLen;  
4097 CK_ULONG firstPieceLen = PLAINTEXT_BUF_SZ / 2;  
4098  
4099 /* error handling is omitted for better readability */
```

```

4100 .
4101 .
4102 C_MessageEncryptInit(hSession, &mechanism, hKey);
4103 /* encrypt message en bloc with given IV */
4104 ulEncryptedDataLen = sizeof(encryptedData[0]);
4105 C_EncryptMessage(hSession,
4106     &gcmParams, sizeof(gcmParams),
4107     &auth[0][0], sizeof(auth[0]),
4108     &data[0][0], sizeof(data[0]),
4109     &encryptedData[0][0], &ulEncryptedDataLen);
4110 /* iv and tag are set now for message */
4111
4112 /* encrypt message in two steps with generated IV */
4113 gcmParams.ivGenerator = CKG_GENERATE;
4114 C_EncryptMessageBegin(hSession,
4115     &gcmParams, sizeof(gcmParams),
4116     &auth[1][0], sizeof(auth[1])
4117 );
4118 /* encrypt first piece */
4119 ulFirstEncryptedDataLen = sizeof(encryptedData[1]);
4120 C_EncryptMessageNext(hSession,
4121     &gcmParams, sizeof(gcmParams),
4122     &data[1][0], firstPieceLen),
4123     &encryptedData[1][0], &ulFirstEncryptedDataLen,
4124     0
4125 );
4126 /* encrypt second piece */
4127 ulEncryptedDataLen = sizeof(encryptedData[1]) - ulFirstEncryptedDataLen;
4128 C_EncryptMessageNext(hSession,
4129     &gcmParams, sizeof(gcmParams),
4130     &data[1][firstPieceLen], sizeof(data[1])-firstPieceLen),
4131     &encryptedData[1][ulFirstEncryptedDataLen], &ulEncryptedDataLen,
4132     CKF_END_OF_MESSAGE
4133 );
4134 /* tag is set now for message */
4135
4136 /* finalize */
4137 C_MessageEncryptFinal(hSession);

```

5.10 Decryption functions

Cryptoki provides the following functions for decrypting data:

5.10.1 C_DecryptInit

```
CK_DECLARE_FUNCTION(CK_RV, C_DecryptInit) (
    CK_SESSION_HANDLE hSession,
    CK_MECHANISM_PTR pMechanism,
    CK_OBJECT_HANDLE hKey
);
```

C_DecryptInit initializes a decryption operation. *hSession* is the session's handle; *pMechanism* points to the decryption mechanism; *hKey* is the handle of the decryption key.

The **CKA_DECRYPT** attribute of the decryption key, which indicates whether the key supports decryption, MUST be CK_TRUE.

After calling **C_DecryptInit**, the application can either call **C_Decrypt** to decrypt data in a single part; or call **C_DecryptUpdate** zero or more times, followed by **C_DecryptFinal**, to decrypt data in multiple parts. The decryption operation is active until the application uses a call to **C_Decrypt** or **C_DecryptFinal** to *actually obtain* the final piece of plaintext. To process additional data (in single or multiple parts), the application MUST call **C_DecryptInit** again.

C_DecryptInit can be called with *pMechanism* set to NULL_PTR to terminate an active decryption operation. If an active operation cannot be cancelled, CKR_OPERATION_CANCEL_FAILED must be returned.

Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID, CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN, CKR_OPERATION_CANCEL_FAILED.

Example: see **C_DecryptFinal**.

5.10.2 C_Decrypt

```
CK_DECLARE_FUNCTION(CK_RV, C_Decrypt) (
    CK_SESSION_HANDLE hSession,
    CK_BYTE_PTR pEncryptedData,
    CK_ULONG ulEncryptedDataLen,
    CK_BYTE_PTR pData,
    CK_ULONG_PTR pulDataLen
);
```

C_Decrypt decrypts encrypted data in a single part. *hSession* is the session's handle; *pEncryptedData* points to the encrypted data; *ulEncryptedDataLen* is the length of the encrypted data; *pData* points to the location that receives the recovered data; *pulDataLen* points to the location that holds the length of the recovered data.

C_Decrypt uses the convention described in Section 5.2 on producing output.

The decryption operation MUST have been initialized with **C_DecryptInit**. A call to **C_Decrypt** always terminates the active decryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the plaintext.

C_Decrypt cannot be used to terminate a multi-part operation, and MUST be called after **C_DecryptInit** without intervening **C_DecryptUpdate** calls.

4186 The ciphertext and plaintext can be in the same place, *i.e.*, it is OK if *pEncryptedData* and *pData* point to
4187 the same location.

4188 If the input ciphertext data cannot be decrypted because it has an inappropriate length, then either
4189 CKR_ENCRYPTED_DATA_INVALID or CKR_ENCRYPTED_DATA_LEN_RANGE may be returned.

4190 For most mechanisms, **C_Decrypt** is equivalent to a sequence of **C_DecryptUpdate** operations followed
4191 by **C_DecryptFinal**.

4192 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4193 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
4194 CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID,
4195 CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
4196 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
4197 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

4198 Example: see **C_DecryptFinal** for an example of similar functions.

4199 5.10.3 C_DecryptUpdate

```
4200 CK_DECLARE_FUNCTION(CK_RV, C_DecryptUpdate) (  
4201     CK_SESSION_HANDLE hSession,  
4202     CK_BYTE_PTR pEncryptedPart,  
4203     CK_ULONG ulEncryptedPartLen,  
4204     CK_BYTE_PTR pPart,  
4205     CK_ULONG_PTR pulPartLen  
4206 );
```

4207 **C_DecryptUpdate** continues a multiple-part decryption operation, processing another encrypted data
4208 part. *hSession* is the session's handle; *pEncryptedPart* points to the encrypted data part;
4209 *ulEncryptedPartLen* is the length of the encrypted data part; *pPart* points to the location that receives the
4210 recovered data part; *pulPartLen* points to the location that holds the length of the recovered data part.

4211 **C_DecryptUpdate** uses the convention described in Section 5.2 on producing output.

4212 The decryption operation MUST have been initialized with **C_DecryptInit**. This function may be called
4213 any number of times in succession. A call to **C_DecryptUpdate** which results in an error other than
4214 CKR_BUFFER_TOO_SMALL terminates the current decryption operation.

4215 The ciphertext and plaintext can be in the same place, *i.e.*, it is OK if *pEncryptedPart* and *pPart* point to
4216 the same location.

4217 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4218 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
4219 CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID,
4220 CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
4221 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
4222 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

4223 Example: See **C_DecryptFinal**.

4224 5.10.4 C_DecryptFinal

```
4225 CK_DECLARE_FUNCTION(CK_RV, C_DecryptFinal) (  
4226     CK_SESSION_HANDLE hSession,  
4227     CK_BYTE_PTR pLastPart,  
4228     CK_ULONG_PTR pulLastPartLen  
4229 );
```

4230 **C_DecryptFinal** finishes a multiple-part decryption operation. *hSession* is the session's handle;
4231 *pLastPart* points to the location that receives the last recovered data part, if any; *pulLastPartLen* points to
4232 the location that holds the length of the last recovered data part.

4233 **C_DecryptFinal** uses the convention described in Section 5.2 on producing output.

4234 The decryption operation MUST have been initialized with **C_DecryptInit**. A call to **C_DecryptFinal**
4235 always terminates the active decryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a
4236 successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the
4237 plaintext.

4238 If the input ciphertext data cannot be decrypted because it has an inappropriate length, then either
4239 CKR_ENCRYPTED_DATA_INVALID or CKR_ENCRYPTED_DATA_LEN_RANGE may be returned.

4240 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4241 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
4242 CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID,
4243 CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
4244 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
4245 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

4246 Example:

```
4247 #define CIPHERTEXT_BUF_SZ 256
4248 #define PLAINTEXT_BUF_SZ 256
4249
4250 CK_ULONG firstEncryptedPieceLen, secondEncryptedPieceLen;
4251 CK_SESSION_HANDLE hSession;
4252 CK_OBJECT_HANDLE hKey;
4253 CK_BYTE iv[8];
4254 CK_MECHANISM mechanism = {
4255     CKM_DES_CBC_PAD, iv, sizeof(iv)
4256 };
4257 CK_BYTE data[PLAINTEXT_BUF_SZ];
4258 CK_BYTE encryptedData[CIPHERTEXT_BUF_SZ];
4259 CK_ULONG ulData1Len, ulData2Len, ulData3Len;
4260 CK_RV rv;
4261
4262 .
4263 .
4264 firstEncryptedPieceLen = 90;
4265 secondEncryptedPieceLen = CIPHERTEXT_BUF_SZ-firstEncryptedPieceLen;
4266 rv = C_DecryptInit(hSession, &mechanism, hKey);
4267 if (rv == CKR_OK) {
4268     /* Decrypt first piece */
4269     ulData1Len = sizeof(data);
4270     rv = C_DecryptUpdate(
4271         hSession,
4272         &encryptedData[0], firstEncryptedPieceLen,
4273         &data[0], &ulData1Len);
4274     if (rv != CKR_OK) {
4275         .
4276         .
4277     }
4278 }
```



```

4279     /* Decrypt second piece */
4280     ulData2Len = sizeof(data)-ulData1Len;
4281     rv = C_DecryptUpdate(
4282         hSession,
4283         &encryptedData[firstEncryptedPieceLen],
4284         secondEncryptedPieceLen,
4285         &data[ulData1Len], &ulData2Len);
4286     if (rv != CKR_OK) {
4287         .
4288         .
4289     }
4290
4291     /* Get last little decrypted bit */
4292     ulData3Len = sizeof(data)-ulData1Len-ulData2Len;
4293     rv = C_DecryptFinal(
4294         hSession,
4295         &data[ulData1Len+ulData2Len], &ulData3Len);
4296     if (rv != CKR_OK) {
4297         .
4298         .
4299     }
4300 }

```

4301 5.11 Message-Based Decryption Functions

4302 Message-based decryption refers to the process of decrypting multiple encrypted messages using the
4303 same decryption mechanism and decryption key. The decryption mechanism can be either an
4304 authenticated encryption with associated data (AEAD) algorithm or a pure encryption algorithm.
4305 Cryptoki provides the following functions for message-based decryption.

4306 5.11.1 C_MessageDecryptInit

```

4307 CK_DECLARE_FUNCTION(CK_RV, C_MessageDecryptInit)(
4308     CK_SESSION_HANDLE hSession,
4309     CK_MECHANISM_PTR pMechanism,
4310     CK_OBJECT_HANDLE hKey
4311 );

```

4312 **C_MessageDecryptInit** initializes a message-based decryption process, preparing a session for one or
4313 more decryption operations that use the same decryption mechanism and decryption key. *hSession* is
4314 the session's handle; *pMechanism* points to the decryption mechanism; *hKey* is the handle of the
4315 decryption key.

4316 The CKA_DECRYPT attribute of the decryption key, which indicates whether the key supports decryption,
4317 MUST be CK_TRUE.

4318 After calling **C_MessageDecryptInit**, the application can either call **C_DecryptMessage** to decrypt an
4319 encrypted message in a single part; or call **C_DecryptMessageBegin**, followed by
4320 **C_DecryptMessageNext** one or more times, to decrypt an encrypted message in multiple parts. This
4321 may be repeated several times. The message-based decryption process is active until the application
4322 uses a call to **C_MessageDecryptFinal** to finish the message-based decryption process.

4323 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
 4324 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
 4325 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
 4326 CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID,
 4327 CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,
 4328 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,
 4329 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
 4330 CKR_OPERATION_CANCEL_FAILED.

4331 5.11.2 C_DecryptMessage

```
4332 CK_DECLARE_FUNCTION(CK_RV, C_DecryptMessage) (
4333     CK_SESSION_HANDLE hSession,
4334     CK_VOID_PTR pParameter,
4335     CK_ULONG ulParameterLen,
4336     CK_BYTE_PTR pAssociatedData,
4337     CK_ULONG ulAssociatedDataLen,
4338     CK_BYTE_PTR pCiphertext,
4339     CK_ULONG ulCiphertextLen,
4340     CK_BYTE_PTR pPlaintext,
4341     CK_ULONG_PTR pulPlaintextLen
4342 );
```

4343 **C_DecryptMessage** decrypts an encrypted message in a single part. *hSession* is the session's handle;
 4344 *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message decryption
 4345 operation; *pAssociatedData* and *ulAssociatedDataLen* specify the associated data for an AEAD
 4346 mechanism; *pCiphertext* points to the encrypted message; *ulCiphertextLen* is the length of the encrypted
 4347 message; *pPlaintext* points to the location that receives the recovered message; *pulPlaintextLen* points to
 4348 the location that holds the length of the recovered message.

4349 Typically, *pParameter* is an initialization vector (IV) or nonce. Unlike the *pParameter* parameter of
 4350 **C_EncryptMessage**, *pParameter* is always an input parameter.

4351 If the decryption mechanism is not AEAD, *pAssociatedData* and *ulAssociatedDataLen* are not used and
 4352 should be set to (NULL, 0).

4353 **C_DecryptMessage** uses the convention described in Section 5.2 on producing output.

4354 The message-based decryption process MUST have been initialized with **C_MessageDecryptInit**. A call
 4355 to **C_DecryptMessage** begins and terminates a message decryption operation.

4356 **C_DecryptMessage** cannot be called in the middle of a multi-part message decryption operation.

4357 The ciphertext and plaintext can be in the same place, i.e., it is OK if *pCiphertext* and *pPlaintext* point to
 4358 the same location.

4359 If the input ciphertext data cannot be decrypted because it has an inappropriate length, then either
 4360 CKR_ENCRYPTED_DATA_INVALID or CKR_ENCRYPTED_DATA_LEN_RANGE may be returned.

4361 If the decryption mechanism is an AEAD algorithm and the authenticity of the associated data or
 4362 ciphertext cannot be verified, then CKR_AEAD_DECRYPT_FAILED is returned.

4363 For most mechanisms, **C_DecryptMessage** is equivalent to **C_DecryptMessageBegin** followed by a
 4364 sequence of **C_DecryptMessageNext** operations.

4365 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
 4366 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
 4367 CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID,
 4368 CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_AEAD_DECRYPT_FAILED,
 4369 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
 4370 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
 4371 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
 4372 CKR_OPERATION_CANCEL_FAILED.

5.11.3 C_DecryptMessageBegin

```
CK_DECLARE_FUNCTION(CK_RV, C_DecryptMessageBegin) (
    CK_SESSION_HANDLE hSession,
    CK_VOID_PTR pParameter,
    CK_ULONG ulParameterLen,
    CK_BYTE_PTR pAssociatedData,
    CK_ULONG ulAssociatedDataLen
);
```

C_DecryptMessageBegin begins a multiple-part message decryption operation. *hSession* is the session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message decryption operation; *pAssociatedData* and *ulAssociatedDataLen* specify the associated data for an AEAD mechanism.

Typically, *pParameter* is an initialization vector (IV) or nonce. Unlike the *pParameter* parameter of **C_EncryptMessageBegin**, *pParameter* is always an input parameter.

If the decryption mechanism is not AEAD, *pAssociatedData* and *ulAssociatedDataLen* are not used and should be set to (NULL, 0).

After calling **C_DecryptMessageBegin**, the application should call **C_DecryptMessageNext** one or more times to decrypt the encrypted message in multiple parts. The message decryption operation is active until the application uses a call to **C_DecryptMessageNext** with flags=CKF_END_OF_MESSAGE to actually obtain the final piece of plaintext. To process additional encrypted messages (in single or multiple parts), the application MUST call **C_DecryptMessage** or **C_DecryptMessageBegin** again.

Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

5.11.4 C_DecryptMessageNext

```
CK_DECLARE_FUNCTION(CK_RV, C_DecryptMessageNext) (
    CK_SESSION_HANDLE hSession,
    CK_BYTE_PTR pCiphertextPart,
    CK_ULONG ulCiphertextPartLen,
    CK_BYTE_PTR pPlaintextPart,
    CK_ULONG_PTR pulPlaintextPartLen,
    CK_FLAGS flags
);
```

C_DecryptMessageNext continues a multiple-part message decryption operation, processing another encrypted message part. *hSession* is the session's handle; *pCiphertextPart* points to the encrypted message part; *ulCiphertextPartLen* is the length of the encrypted message part; *pPlaintextPart* points to the location that receives the recovered message part; *pulPlaintextPartLen* points to the location that holds the length of the recovered message part; flags is set to 0 if there is more ciphertext data to follow, or set to CKF_END_OF_MESSAGE if this is the last ciphertext part.

C_DecryptMessageNext uses the convention described in Section 5.2 on producing output.

The message decryption operation MUST have been started with **C_DecryptMessageBegin**. This function may be called any number of times in succession. A call to **C_DecryptMessageNext** with flags=0 which results in an error other than CKR_BUFFER_TOO_SMALL terminates the current message decryption operation. A call to **C_DecryptMessageNext** with flags=CKF_END_OF_MESSAGE always terminates the active message decryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful call (i.e., one which returns CKR_OK) to determine the length of the buffer needed to hold the plaintext.

The ciphertext and plaintext can be in the same place, i.e., it is OK if *pCiphertextPart* and *pPlaintextPart* point to the same location.

4424 Although the last **C_DecryptMessageNext** call ends the decryption of a message, it does not finish the
4425 message-based decryption process. Additional **C_DecryptMessage** or **C_DecryptMessageBegin** and
4426 **C_DecryptMessageNext** calls may be made on the session.

4427 If the input ciphertext data cannot be decrypted because it has an inappropriate length, then either
4428 CKR_ENCRYPTED_DATA_INVALID or CKR_ENCRYPTED_DATA_LEN_RANGE may be returned by
4429 the last **C_DecryptMessageNext** call.

4430 If the decryption mechanism is an AEAD algorithm and the authenticity of the associated data or
4431 ciphertext cannot be verified, then CKR_AEAD_DECRYPT_FAILED is returned by the last
4432 **C_DecryptMessageNext** call.

4433 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4434 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
4435 CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID,
4436 CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_AEAD_DECRYPT_FAILED,
4437 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4438 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
4439 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

4440 5.11.5 C_MessageDecryptFinal

```
4441 CK_DECLARE_FUNCTION(CK_RV, C_MessageDecryptFinal) (  
4442     CK_SESSION_HANDLE hSession  
4443 );
```

4444 **C_MessageDecryptFinal** finishes a message-based decryption process. *hSession* is the session's
4445 handle.

4446 The message-based decryption process MUST have been initialized with **C_MessageDecryptInit**.

4447 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4448 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4449 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4450 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
4451 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

4452 5.12 Message digesting functions

4453 Cryptoki provides the following functions for digesting data:

4454 5.12.1 C_DigestInit

```
4455 CK_DECLARE_FUNCTION(CK_RV, C_DigestInit) (  
4456     CK_SESSION_HANDLE hSession,  
4457     CK_MECHANISM_PTR pMechanism  
4458 );
```

4459 **C_DigestInit** initializes a message-digesting operation. *hSession* is the session's handle; *pMechanism*
4460 points to the digesting mechanism.

4461 After calling **C_DigestInit**, the application can either call **C_Digest** to digest data in a single part; or call
4462 **C_DigestUpdate** zero or more times, followed by **C_DigestFinal**, to digest data in multiple parts. The
4463 message-digesting operation is active until the application uses a call to **C_Digest** or **C_DigestFinal** to
4464 *actually obtain* the message digest. To process additional data (in single or multiple parts), the
4465 application MUST call **C_DigestInit** again.

4466 **C_DigestInit** can be called with *pMechanism* set to NULL_PTR to terminate an active message-digesting
4467 operation. If an operation has been initialized and it cannot be cancelled,
4468 CKR_OPERATION_CANCEL_FAILED must be returned.

4469 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4470 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,

4471 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
 4472 CKR_HOST_MEMORY, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID,
 4473 CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED,
 4474 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
 4475 CKR_OPERATION_CANCEL_FAILED.

4476 Example: see **C_DigestFinal**.

4477 5.12.2 C_Digest

```
4478 CK_DECLARE_FUNCTION(CK_RV, C_Digest) (
4479     CK_SESSION_HANDLE hSession,
4480     CK_BYTE_PTR pData,
4481     CK_ULONG ulDataLen,
4482     CK_BYTE_PTR pDigest,
4483     CK_ULONG_PTR pulDigestLen
4484 );
```

4485 **C_Digest** digests data in a single part. *hSession* is the session's handle, *pData* points to the data;
 4486 *ulDataLen* is the length of the data; *pDigest* points to the location that receives the message digest;
 4487 *pulDigestLen* points to the location that holds the length of the message digest.

4488 **C_Digest** uses the convention described in Section 5.2 on producing output.

4489 The digest operation MUST have been initialized with **C_DigestInit**. A call to **C_Digest** always
 4490 terminates the active digest operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful
 4491 call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the message
 4492 digest.

4493 **C_Digest** cannot be used to terminate a multi-part operation, and MUST be called after **C_DigestInit**
 4494 without intervening **C_DigestUpdate** calls.

4495 The input data and digest output can be in the same place, *i.e.*, it is OK if *pData* and *pDigest* point to the
 4496 same location.

4497 **C_Digest** is equivalent to a sequence of **C_DigestUpdate** operations followed by **C_DigestFinal**.

4498 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
 4499 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
 4500 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
 4501 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
 4502 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

4503 Example: see **C_DigestFinal** for an example of similar functions.

4504 5.12.3 C_DigestUpdate

```
4505 CK_DECLARE_FUNCTION(CK_RV, C_DigestUpdate) (
4506     CK_SESSION_HANDLE hSession,
4507     CK_BYTE_PTR pPart,
4508     CK_ULONG ulPartLen
4509 );
```

4510 **C_DigestUpdate** continues a multiple-part message-digesting operation, processing another data part.
 4511 *hSession* is the session's handle, *pPart* points to the data part; *ulPartLen* is the length of the data part.

4512 The message-digesting operation MUST have been initialized with **C_DigestInit**. Calls to this function
 4513 and **C_DigestKey** may be interspersed any number of times in any order. A call to **C_DigestUpdate**
 4514 which results in an error terminates the current digest operation.

4515 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
 4516 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
 4517 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
 4518 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
 4519 CKR_SESSION_HANDLE_INVALID.

4520 Example: see **C_DigestFinal**.

4521 5.12.4 C_DigestKey

```
4522 CK_DECLARE_FUNCTION(CK_RV, C_DigestKey) (  
4523     CK_SESSION_HANDLE hSession,  
4524     CK_OBJECT_HANDLE hKey  
4525 );
```

4526 **C_DigestKey** continues a multiple-part message-digesting operation by digesting the value of a secret
4527 key. *hSession* is the session's handle; *hKey* is the handle of the secret key to be digested.

4528 The message-digesting operation MUST have been initialized with **C_DigestInit**. Calls to this function
4529 and **C_DigestUpdate** may be interspersed any number of times in any order.

4530 If the value of the supplied key cannot be digested purely for some reason related to its length,
4531 **C_DigestKey** should return the error code CKR_KEY_SIZE_RANGE.

4532 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
4533 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
4534 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_HANDLE_INVALID,
4535 CKR_KEY_INDIGESTIBLE, CKR_KEY_SIZE_RANGE, CKR_OK,
4536 CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

4537 Example: see **C_DigestFinal**.

4538 5.12.5 C_DigestFinal

```
4539 CK_DECLARE_FUNCTION(CK_RV, C_DigestFinal) (  
4540     CK_SESSION_HANDLE hSession,  
4541     CK_BYTE_PTR pDigest,  
4542     CK_ULONG_PTR pulDigestLen  
4543 );
```

4544 **C_DigestFinal** finishes a multiple-part message-digesting operation, returning the message digest.
4545 *hSession* is the session's handle; *pDigest* points to the location that receives the message digest;
4546 *pulDigestLen* points to the location that holds the length of the message digest.

4547 **C_DigestFinal** uses the convention described in Section 5.2 on producing output.

4548 The digest operation MUST have been initialized with **C_DigestInit**. A call to **C_DigestFinal** always
4549 terminates the active digest operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful
4550 call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the message
4551 digest.

4552 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4553 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
4554 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
4555 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
4556 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

4557 Example:

```
4558 CK_SESSION_HANDLE hSession;  
4559 CK_MECHANISM mechanism = {  
4560     CKM_MD5, NULL_PTR, 0  
4561 };  
4562 CK_BYTE data[] = {...};  
4563 CK_BYTE digest[16];  
4564 CK_ULONG ulDigestLen;  
4565 CK_RV rv;
```



```

4566
4567 .
4568 .
4569 rv = C_DigestInit(hSession, &mechanism);
4570 if (rv != CKR_OK) {
4571     .
4572     .
4573 }
4574
4575 rv = C_DigestUpdate(hSession, data, sizeof(data));
4576 if (rv != CKR_OK) {
4577     .
4578     .
4579 }
4580
4581 rv = C_DigestKey(hSession, hKey);
4582 if (rv != CKR_OK) {
4583     .
4584     .
4585 }
4586
4587 ulDigestLen = sizeof(digest);
4588 rv = C_DigestFinal(hSession, digest, &ulDigestLen);
4589 .
4590 .

```

4591 5.13 Signing and MACing functions

4592 Cryptoki provides the following functions for signing data (for the purposes of Cryptoki, these operations
4593 also encompass message authentication codes).

4594 5.13.1 C_SignInit

```

4595 CK_DECLARE_FUNCTION(CK_RV, C_SignInit)(
4596     CK_SESSION_HANDLE hSession,
4597     CK_MECHANISM_PTR pMechanism,
4598     CK_OBJECT_HANDLE hKey
4599 );

```

4600 **C_SignInit** initializes a signature operation, where the signature is an appendix to the data. *hSession* is
4601 the session's handle; *pMechanism* points to the signature mechanism; *hKey* is the handle of the signature
4602 key.

4603 The **CKA_SIGN** attribute of the signature key, which indicates whether the key supports signatures with
4604 appendix, MUST be CK_TRUE.

4605 After calling **C_SignInit**, the application can either call **C_Sign** to sign in a single part; or call
4606 **C_SignUpdate** one or more times, followed by **C_SignFinal**, to sign data in multiple parts. The signature
4607 operation is active until the application uses a call to **C_Sign** or **C_SignFinal** to *actually obtain* the
4608 signature. To process additional data (in single or multiple parts), the application MUST call **C_SignInit**
4609 again.

4610 **C_SignInit** can be called with *pMechanism* set to `NULL_PTR` to terminate an active signature operation.
4611 If an operation has been initialized and it cannot be cancelled, `CKR_OPERATION_CANCEL_FAILED`
4612 must be returned.

4613 Return values: `CKR_ARGUMENTS_BAD`, `CKR_CRYPTOKI_NOT_INITIALIZED`,
4614 `CKR_DEVICE_ERROR`, `CKR_DEVICE_MEMORY`, `CKR_DEVICE_REMOVED`,
4615 `CKR_FUNCTION_CANCELED`, `CKR_FUNCTION_FAILED`, `CKR_GENERAL_ERROR`,
4616 `CKR_HOST_MEMORY`, `CKR_KEY_FUNCTION_NOT_PERMITTED`, `CKR_KEY_HANDLE_INVALID`,
4617 `CKR_KEY_SIZE_RANGE`, `CKR_KEY_TYPE_INCONSISTENT`, `CKR_MECHANISM_INVALID`,
4618 `CKR_MECHANISM_PARAM_INVALID`, `CKR_OK`, `CKR_OPERATION_ACTIVE`, `CKR_PIN_EXPIRED`,
4619 `CKR_SESSION_CLOSED`, `CKR_SESSION_HANDLE_INVALID`, `CKR_USER_NOT_LOGGED_IN`,
4620 `CKR_OPERATION_CANCEL_FAILED`.

4621 Example: see **C_SignFinal**.

4622 5.13.2 C_Sign

```
4623 CK_DECLARE_FUNCTION(CK_RV, C_Sign) (  
4624     CK_SESSION_HANDLE hSession,  
4625     CK_BYTE_PTR pData,  
4626     CK_ULONG ulDataLen,  
4627     CK_BYTE_PTR pSignature,  
4628     CK_ULONG_PTR pulSignatureLen  
4629 );
```

4630 **C_Sign** signs data in a single part, where the signature is an appendix to the data. *hSession* is the
4631 session's handle; *pData* points to the data; *ulDataLen* is the length of the data; *pSignature* points to the
4632 location that receives the signature; *pulSignatureLen* points to the location that holds the length of the
4633 signature.

4634 **C_Sign** uses the convention described in Section 5.2 on producing output.

4635 The signing operation **MUST** have been initialized with **C_SignInit**. A call to **C_Sign** always terminates
4636 the active signing operation unless it returns `CKR_BUFFER_TOO_SMALL` or is a successful call (*i.e.*,
4637 one which returns `CKR_OK`) to determine the length of the buffer needed to hold the signature.

4638 **C_Sign** cannot be used to terminate a multi-part operation, and **MUST** be called after **C_SignInit** without
4639 intervening **C_SignUpdate** calls.

4640 For most mechanisms, **C_Sign** is equivalent to a sequence of **C_SignUpdate** operations followed by
4641 **C_SignFinal**.

4642 Return values: `CKR_ARGUMENTS_BAD`, `CKR_BUFFER_TOO_SMALL`,
4643 `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DATA_INVALID`, `CKR_DATA_LEN_RANGE`,
4644 `CKR_DEVICE_ERROR`, `CKR_DEVICE_MEMORY`, `CKR_DEVICE_REMOVED`,
4645 `CKR_FUNCTION_CANCELED`, `CKR_FUNCTION_FAILED`, `CKR_GENERAL_ERROR`,
4646 `CKR_HOST_MEMORY`, `CKR_OK`, `CKR_OPERATION_NOT_INITIALIZED`, `CKR_SESSION_CLOSED`,
4647 `CKR_SESSION_HANDLE_INVALID`, `CKR_USER_NOT_LOGGED_IN`, `CKR_FUNCTION_REJECTED`,
4648 `CKR_TOKEN_RESOURCE_EXCEEDED`.

4649 Example: see **C_SignFinal** for an example of similar functions.

4650 5.13.3 C_SignUpdate

```
4651 CK_DECLARE_FUNCTION(CK_RV, C_SignUpdate) (  
4652     CK_SESSION_HANDLE hSession,  
4653     CK_BYTE_PTR pPart,  
4654     CK_ULONG ulPartLen  
4655 );
```

4656 **C_SignUpdate** continues a multiple-part signature operation, processing another data part. *hSession* is
4657 the session's handle, *pPart* points to the data part; *ulPartLen* is the length of the data part.

4658 The signature operation MUST have been initialized with **C_SignInit**. This function may be called any
4659 number of times in succession. A call to **C_SignUpdate** which results in an error terminates the current
4660 signature operation.

4661 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4662 CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
4663 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
4664 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
4665 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
4666 CKR_TOKEN_RESOURCE_EXCEEDED.

4667 Example: see **C_SignFinal**.

4668 5.13.4 C_SignFinal

```
4669 CK_DECLARE_FUNCTION(CK_RV, C_SignFinal)(  
4670     CK_SESSION_HANDLE hSession,  
4671     CK_BYTE_PTR pSignature,  
4672     CK_ULONG_PTR pulSignatureLen  
4673 );
```

4674 **C_SignFinal** finishes a multiple-part signature operation, returning the signature. *hSession* is the
4675 session's handle; *pSignature* points to the location that receives the signature; *pulSignatureLen* points to
4676 the location that holds the length of the signature.

4677 **C_SignFinal** uses the convention described in Section 5.2 on producing output.

4678 The signing operation MUST have been initialized with **C_SignInit**. A call to **C_SignFinal** always
4679 terminates the active signing operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful
4680 call (i.e., one which returns CKR_OK) to determine the length of the buffer needed to hold the signature.

4681 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4682 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR,
4683 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED,
4684 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
4685 CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
4686 CKR_USER_NOT_LOGGED_IN, CKR_FUNCTION_REJECTED,
4687 CKR_TOKEN_RESOURCE_EXCEEDED.

4688 Example:

```
4689 CK_SESSION_HANDLE hSession;  
4690 CK_OBJECT_HANDLE hKey;  
4691 CK_MECHANISM mechanism = {  
4692     CKM_DES_MAC, NULL_PTR, 0  
4693 };  
4694 CK_BYTE data[] = {...};  
4695 CK_BYTE mac[4];  
4696 CK_ULONG ulMacLen;  
4697 CK_RV rv;  
4698  
4699 .  
4700 .  
4701 rv = C_SignInit(hSession, &mechanism, hKey);  
4702 if (rv == CKR_OK) {  
4703     rv = C_SignUpdate(hSession, data, sizeof(data));  
4704     .
```

```

4705 .
4706     ulMacLen = sizeof(mac);
4707     rv = C_SignFinal(hSession, mac, &ulMacLen);
4708 .
4709 .
4710 }

```

4711 5.13.5 C_SignRecoverInit

```

4712 CK_DECLARE_FUNCTION(CK_RV, C_SignRecoverInit) (
4713     CK_SESSION_HANDLE hSession,
4714     CK_MECHANISM_PTR pMechanism,
4715     CK_OBJECT_HANDLE hKey
4716 );

```

4717 **C_SignRecoverInit** initializes a signature operation, where the data can be recovered from the signature.
4718 *hSession* is the session's handle; *pMechanism* points to the structure that specifies the signature
4719 mechanism; *hKey* is the handle of the signature key.

4720 The **CKA_SIGN_RECOVER** attribute of the signature key, which indicates whether the key supports
4721 signatures where the data can be recovered from the signature, MUST be CK_TRUE.

4722 After calling **C_SignRecoverInit**, the application may call **C_SignRecover** to sign in a single part. The
4723 signature operation is active until the application uses a call to **C_SignRecover** to *actually obtain* the
4724 signature. To process additional data in a single part, the application MUST call **C_SignRecoverInit**
4725 again.

4726 **C_SignRecoverInit** can be called with *pMechanism* set to NULL_PTR to terminate an active signature
4727 with data recovery operation. If an active operation has been initialized and it cannot be cancelled,
4728 CKR_OPERATION_CANCEL_FAILED must be returned.

4729 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4730 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4731 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4732 CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID,
4733 CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,
4734 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,
4735 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
4736 CKR_OPERATION_CANCEL_FAILED.

4737 Example: see **C_SignRecover**.

4738 5.13.6 C_SignRecover

```

4739 CK_DECLARE_FUNCTION(CK_RV, C_SignRecover) (
4740     CK_SESSION_HANDLE hSession,
4741     CK_BYTE_PTR pData,
4742     CK_ULONG ulDataLen,
4743     CK_BYTE_PTR pSignature,
4744     CK_ULONG_PTR pulSignatureLen
4745 );

```

4746 **C_SignRecover** signs data in a single operation, where the data can be recovered from the signature.
4747 *hSession* is the session's handle; *pData* points to the data; *ulDataLen* is the length of the data;
4748 *pSignature* points to the location that receives the signature; *pulSignatureLen* points to the location that
4749 holds the length of the signature.

4750 **C_SignRecover** uses the convention described in Section 5.2 on producing output.

4751 The signing operation MUST have been initialized with **C_SignRecoverInit**. A call to **C_SignRecover**
4752 always terminates the active signing operation unless it returns CKR_BUFFER_TOO_SMALL or is a

4753 successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the
4754 signature.

4755 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4756 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE,
4757 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4758 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4759 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
4760 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
4761 CKR_TOKEN_RESOURCE_EXCEEDED.

4762 Example:

```
4763 CK_SESSION_HANDLE hSession;  
4764 CK_OBJECT_HANDLE hKey;  
4765 CK_MECHANISM mechanism = {  
4766     CKM_RSA_9796, NULL_PTR, 0  
4767 };  
4768 CK_BYTE data[] = {...};  
4769 CK_BYTE signature[128];  
4770 CK_ULONG ulSignatureLen;  
4771 CK_RV rv;  
4772  
4773 .  
4774 .  
4775 rv = C_SignRecoverInit(hSession, &mechanism, hKey);  
4776 if (rv == CKR_OK) {  
4777     ulSignatureLen = sizeof(signature);  
4778     rv = C_SignRecover(  
4779         hSession, data, sizeof(data), signature, &ulSignatureLen);  
4780     if (rv == CKR_OK) {  
4781         .  
4782         .  
4783     }  
4784 }
```

4785 Functions for verifying signatures and MACs

4786 5.14 Message-Based Signing and MACing Functions

4787 Message-based signature refers to the process of signing multiple messages using the same signature
4788 mechanism and signature key.

4789 Cryptoki provides the following functions for for signing messages (for the purposes of Cryptoki, these
4790 operations also encompass message authentication codes).

4791 5.14.1 C_MessageSignInit

```
4792 CK_DECLARE_FUNCTION(CK_RV, C_MessageSignInit) (  
4793     CK_SESSION_HANDLE hSession,  
4794     CK_MECHANISM_PTR pMechanism,  
4795     CK_OBJECT_HANDLE hKey
```

4796);

4797 **C_MessageSignInit** initializes a message-based signature process, preparing a session for one or more
4798 signature operations (where the signature is an appendix to the data) that use the same signature
4799 mechanism and signature key. *hSession* is the session's handle; *pMechanism* points to the signature
4800 mechanism; *hKey* is the handle of the signature key.

4801 The **CKA_SIGN** attribute of the signature key, which indicates whether the key supports signatures with
4802 appendix, MUST be CK_TRUE.

4803 After calling **C_MessageSignInit**, the application can either call **C_SignMessage** to sign a message in a
4804 single part; or call **C_SignMessageBegin**, followed by **C_SignMessageNext** one or more times, to sign
4805 a message in multiple parts. This may be repeated several times. The message-based signature process
4806 is active until the application calls **C_MessageSignFinal** to finish the message-based signature process.

4807 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4808 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4809 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4810 CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID,
4811 CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,
4812 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,
4813 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

4814 5.14.2 C_SignMessage

```
4815 CK_DECLARE_FUNCTION(CK_RV, C_SignMessage) (  
4816     CK_SESSION_HANDLE hSession,  
4817     CK_VOID_PTR pParameter,  
4818     CK_ULONG ulParameterLen,  
4819     CK_BYTE_PTR pData,  
4820     CK_ULONG ulDataLen,  
4821     CK_BYTE_PTR pSignature,  
4822     CK_ULONG_PTR pulSignatureLen  
4823 );
```

4824 **C_SignMessage** signs a message in a single part, where the signature is an appendix to the message.

4825 **C_MessageSignInit** must previously been called on the session. *hSession* is the session's handle;
4826 *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message signature
4827 operation; *pData* points to the data; *ulDataLen* is the length of the data; *pSignature* points to the location
4828 that receives the signature; *pulSignatureLen* points to the location that holds the length of the signature.

4829 Depending on the mechanism parameter passed to **C_MessageSignInit**, *pParameter* may be either an
4830 input or an output parameter.

4831 **C_SignMessage** uses the convention described in Section 5.2 on producing output.

4832 The message-based signing process MUST have been initialized with **C_MessageSignInit**. A call to
4833 **C_SignMessage** begins and terminates a message signing operation unless it returns
4834 CKR_BUFFER_TOO_SMALL to determine the length of the buffer needed to hold the signature, or is a
4835 successful call (i.e., one which returns CKR_OK).

4836 **C_SignMessage** cannot be called in the middle of a multi-part message signing operation.

4837 **C_SignMessage** does not finish the message-based signing process. Additional **C_SignMessage** or
4838 **C_SignMessageBegin** and **C_SignMessageNext** calls may be made on the session.

4839 For most mechanisms, **C_SignMessage** is equivalent to **C_SignMessageBegin** followed by a sequence
4840 of **C_SignMessageNext** operations.

4841 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4842 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE,
4843 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,

4844 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4845 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
4846 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN, CKR_FUNCTION_REJECTED,
4847 CKR_TOKEN_RESOURCE_EXCEEDED.

4848 5.14.3 C_SignMessageBegin

```
4849 CK_DECLARE_FUNCTION(CK_RV, C_SignMessageBegin) (  
4850     CK_SESSION_HANDLE hSession,  
4851     CK_VOID_PTR pParameter,  
4852     CK_ULONG ulParameterLen  
4853 );
```

4854 **C_SignMessageBegin** begins a multiple-part message signature operation, where the signature is an
4855 appendix to the message. **C_MessageSignInit** must previously been called on the session. *hSession* is
4856 the session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for
4857 the message signature operation.

4858 Depending on the mechanism parameter passed to **C_MessageSignInit**, *pParameter* may be either an
4859 input or an output parameter.

4860 After calling **C_SignMessageBegin**, the application should call **C_SignMessageNext** one or more times
4861 to sign the message in multiple parts. The message signature operation is active until the application
4862 uses a call to **C_SignMessageNext** with a non-NULL *pulSignatureLen* to actually obtain the signature.
4863 To process additional messages (in single or multiple parts), the application MUST call **C_SignMessage**
4864 or **C_SignMessageBegin** again.

4865 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4866 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4867 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4868 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,
4869 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
4870 CKR_TOKEN_RESOURCE_EXCEEDED.

4871 5.14.4 C_SignMessageNext

```
4872 CK_DECLARE_FUNCTION(CK_RV, C_SignMessageNext) (  
4873     CK_SESSION_HANDLE hSession,  
4874     CK_VOID_PTR pParameter,  
4875     CK_ULONG ulParameterLen,  
4876     CK_BYTE_PTR pDataPart,  
4877     CK_ULONG ulDataPartLen,  
4878     CK_BYTE_PTR pSignature,  
4879     CK_ULONG_PTR pulSignatureLen  
4880 );
```

4881 **C_SignMessageNext** continues a multiple-part message signature operation, processing another data
4882 part, or finishes a multiple-part message signature operation, returning the signature. *hSession* is the
4883 session's handle, *pDataPart* points to the data part; *pParameter* and *ulParameterLen* specify any
4884 mechanism-specific parameters for the message signature operation; *ulDataPartLen* is the length of the
4885 data part; *pSignature* points to the location that receives the signature; *pulSignatureLen* points to the
4886 location that holds the length of the signature.

4887 The *pulSignatureLen* argument is set to NULL if there is more data part to follow, or set to a non-NULL
4888 value (to receive the signature length) if this is the last data part.

4889 **C_SignMessageNext** uses the convention described in Section 5.2 on producing output.

4890 The message signing operation MUST have been started with **C_SignMessageBegin**. This function may
4891 be called any number of times in succession. A call to **C_SignMessageNext** with a NULL
4892 *pulSignatureLen* which results in an error terminates the current message signature operation. A call to
4893 **C_SignMessageNext** with a non-NULL *pulSignatureLen* always terminates the active message signing
4894 operation unless it returns CKR_BUFFER_TOO_SMALL to determine the length of the buffer needed to
4895 hold the signature, or is a successful call (i.e., one which returns CKR_OK).

4896 Although the last **C_SignMessageNext** call ends the signing of a message, it does not finish the
4897 message-based signing process. Additional **C_SignMessage** or **C_SignMessageBegin** and
4898 **C_SignMessageNext** calls may be made on the session.

4899 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4900 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR,
4901 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED,
4902 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
4903 CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
4904 CKR_USER_NOT_LOGGED_IN, CKR_FUNCTION_REJECTED,
4905 CKR_TOKEN_RESOURCE_EXCEEDED.

4906 5.14.5 C_MessageSignFinal

```
4907 CK_DECLARE_FUNCTION(CK_RV, C_MessageSignFinal) (  
4908     CK_SESSION_HANDLE hSession  
4909 );
```

4910 **C_MessageSignFinal** finishes a message-based signing process. *hSession* is the session's handle.

4911 The message-based signing process MUST have been initialized with **C_MessageSignInit**.

4912 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4913 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4914 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4915 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
4916 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN, CKR_FUNCTION_REJECTED,
4917 CKR_TOKEN_RESOURCE_EXCEEDED.

4918 5.15 Functions for Verifying Signatures and MACs

4919 Cryptoki provides the following functions for verifying signatures on data (for the purposes of Cryptoki,
4920 these operations also encompass message authentication codes):

4921 5.15.1 C_VerifyInit

```
4922 CK_DECLARE_FUNCTION(CK_RV, C_VerifyInit) (  
4923     CK_SESSION_HANDLE hSession,  
4924     CK_MECHANISM_PTR pMechanism,  
4925     CK_OBJECT_HANDLE hKey  
4926 );
```

4927 **C_VerifyInit** initializes a verification operation, where the signature is an appendix to the data. *hSession*
4928 is the session's handle; *pMechanism* points to the structure that specifies the verification mechanism;
4929 *hKey* is the handle of the verification key.

4930 The **CKA_VERIFY** attribute of the verification key, which indicates whether the key supports verification
4931 where the signature is an appendix to the data, MUST be CK_TRUE.

4932 After calling **C_VerifyInit**, the application can either call **C_Verify** to verify a signature on data in a single
4933 part; or call **C_VerifyUpdate** one or more times, followed by **C_VerifyFinal**, to verify a signature on data
4934 in multiple parts. The verification operation is active until the application calls **C_Verify** or **C_VerifyFinal**.
4935 To process additional data (in single or multiple parts), the application MUST call **C_VerifyInit** again.

4936 **C_VerifyInit** can be called with *pMechanism* set to `NULL_PTR` to terminate an active verification
4937 operation. If an active operation has been initialized and it cannot be cancelled,
4938 `CKR_OPERATION_CANCEL_FAILED` must be returned.

4939 Return values: `CKR_ARGUMENTS_BAD`, `CKR_CRYPTOKI_NOT_INITIALIZED`,
4940 `CKR_DEVICE_ERROR`, `CKR_DEVICE_MEMORY`, `CKR_DEVICE_REMOVED`,
4941 `CKR_FUNCTION_CANCELED`, `CKR_FUNCTION_FAILED`, `CKR_GENERAL_ERROR`,
4942 `CKR_HOST_MEMORY`, `CKR_KEY_FUNCTION_NOT_PERMITTED`, `CKR_KEY_HANDLE_INVALID`,
4943 `CKR_KEY_SIZE_RANGE`, `CKR_KEY_TYPE_INCONSISTENT`, `CKR_MECHANISM_INVALID`,
4944 `CKR_MECHANISM_PARAM_INVALID`, `CKR_OK`, `CKR_OPERATION_ACTIVE`, `CKR_PIN_EXPIRED`,
4945 `CKR_SESSION_CLOSED`, `CKR_SESSION_HANDLE_INVALID`, `CKR_USER_NOT_LOGGED_IN`,
4946 `CKR_OPERATION_CANCEL_FAILED`.

4947 Example: see **C_VerifyFinal**.

4948 5.15.2 C_Verify

```
4949 CK_DECLARE_FUNCTION(CK_RV, C_Verify)(  
4950     CK_SESSION_HANDLE hSession,  
4951     CK_BYTE_PTR pData,  
4952     CK_ULONG ulDataLen,  
4953     CK_BYTE_PTR pSignature,  
4954     CK_ULONG ulSignatureLen  
4955 );
```

4956 **C_Verify** verifies a signature in a single-part operation, where the signature is an appendix to the data.
4957 *hSession* is the session's handle; *pData* points to the data; *ulDataLen* is the length of the data;
4958 *pSignature* points to the signature; *ulSignatureLen* is the length of the signature.

4959 The verification operation **MUST** have been initialized with **C_VerifyInit**. A call to **C_Verify** always
4960 terminates the active verification operation.

4961 A successful call to **C_Verify** should return either the value `CKR_OK` (indicating that the supplied
4962 signature is valid) or `CKR_SIGNATURE_INVALID` (indicating that the supplied signature is invalid). If the
4963 signature can be seen to be invalid purely on the basis of its length, then
4964 `CKR_SIGNATURE_LEN_RANGE` should be returned. In any of these cases, the active signing operation
4965 is terminated.

4966 **C_Verify** cannot be used to terminate a multi-part operation, and **MUST** be called after **C_VerifyInit**
4967 without intervening **C_VerifyUpdate** calls.

4968 For most mechanisms, **C_Verify** is equivalent to a sequence of **C_VerifyUpdate** operations followed by
4969 **C_VerifyFinal**.

4970 Return values: `CKR_ARGUMENTS_BAD`, `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DATA_INVALID`,
4971 `CKR_DATA_LEN_RANGE`, `CKR_DEVICE_ERROR`, `CKR_DEVICE_MEMORY`,
4972 `CKR_DEVICE_REMOVED`, `CKR_FUNCTION_CANCELED`, `CKR_FUNCTION_FAILED`,
4973 `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`, `CKR_OPERATION_NOT_INITIALIZED`,
4974 `CKR_SESSION_CLOSED`, `CKR_SESSION_HANDLE_INVALID`, `CKR_SIGNATURE_INVALID`,
4975 `CKR_SIGNATURE_LEN_RANGE`, `CKR_TOKEN_RESOURCE_EXCEEDED`.

4976 Example: see **C_VerifyFinal** for an example of similar functions.

4977 5.15.3 C_VerifyUpdate

```
4978 CK_DECLARE_FUNCTION(CK_RV, C_VerifyUpdate)(  
4979     CK_SESSION_HANDLE hSession,  
4980     CK_BYTE_PTR pPart,  
4981     CK_ULONG ulPartLen  
4982 );
```

4983 **C_VerifyUpdate** continues a multiple-part verification operation, processing another data part. *hSession*
4984 is the session's handle, *pPart* points to the data part; *ulPartLen* is the length of the data part.

4985 The verification operation MUST have been initialized with **C_VerifyInit**. This function may be called any
4986 number of times in succession. A call to **C_VerifyUpdate** which results in an error terminates the current
4987 verification operation.

4988 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4989 CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
4990 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
4991 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
4992 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
4993 CKR_TOKEN_RESOURCE_EXCEEDED.

4994 Example: see **C_VerifyFinal**.

4995 5.15.4 C_VerifyFinal

```
4996 CK_DECLARE_FUNCTION(CK_RV, C_VerifyFinal)(  
4997     CK_SESSION_HANDLE hSession,  
4998     CK_BYTE_PTR pSignature,  
4999     CK_ULONG ulSignatureLen  
5000 );
```

5001 **C_VerifyFinal** finishes a multiple-part verification operation, checking the signature. *hSession* is the
5002 session's handle; *pSignature* points to the signature; *ulSignatureLen* is the length of the signature.

5003 The verification operation MUST have been initialized with **C_VerifyInit**. A call to **C_VerifyFinal** always
5004 terminates the active verification operation.

5005 A successful call to **C_VerifyFinal** should return either the value CKR_OK (indicating that the supplied
5006 signature is valid) or CKR_SIGNATURE_INVALID (indicating that the supplied signature is invalid). If the
5007 signature can be seen to be invalid purely on the basis of its length, then
5008 CKR_SIGNATURE_LEN_RANGE should be returned. In any of these cases, the active verifying
5009 operation is terminated.

5010 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
5011 CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
5012 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
5013 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
5014 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SIGNATURE_INVALID,
5015 CKR_SIGNATURE_LEN_RANGE, CKR_TOKEN_RESOURCE_EXCEEDED.

5016 Example:

```
5017 CK_SESSION_HANDLE hSession;  
5018 CK_OBJECT_HANDLE hKey;  
5019 CK_MECHANISM mechanism = {  
5020     CKM_DES_MAC, NULL_PTR, 0  
5021 };  
5022 CK_BYTE data[] = {...};  
5023 CK_BYTE mac[4];  
5024 CK_RV rv;  
5025  
5026 .  
5027 .  
5028 rv = C_VerifyInit(hSession, &mechanism, hKey);  
5029 if (rv == CKR_OK) {  
5030     rv = C_VerifyUpdate(hSession, data, sizeof(data));  
5031     .
```



```

5032 .
5033 rv = C_VerifyFinal(hSession, mac, sizeof(mac));
5034 .
5035 .
5036 }

```

5037 5.15.5 C_VerifyRecoverInit

```

5038 CK_DECLARE_FUNCTION(CK_RV, C_VerifyRecoverInit) (
5039     CK_SESSION_HANDLE hSession,
5040     CK_MECHANISM_PTR pMechanism,
5041     CK_OBJECT_HANDLE hKey
5042 );

```

5043 **C_VerifyRecoverInit** initializes a signature verification operation, where the data is recovered from the
5044 signature. *hSession* is the session's handle; *pMechanism* points to the structure that specifies the
5045 verification mechanism; *hKey* is the handle of the verification key.

5046 The **CKA_VERIFY_RECOVER** attribute of the verification key, which indicates whether the key supports
5047 verification where the data is recovered from the signature, **MUST** be **CK_TRUE**.

5048 After calling **C_VerifyRecoverInit**, the application may call **C_VerifyRecover** to verify a signature on
5049 data in a single part. The verification operation is active until the application uses a call to
5050 **C_VerifyRecover** to *actually obtain* the recovered message.

5051 **C_VerifyRecoverInit** can be called with *pMechanism* set to **NULL_PTR** to terminate an active verification
5052 with data recovery operation. If an active operations has been initialized and it cannot be cancelled,
5053 **CKR_OPERATION_CANCEL_FAILED** must be returned.

5054 Return values: **CKR_ARGUMENTS_BAD**, **CKR_CRYPTOKI_NOT_INITIALIZED**,
5055 **CKR_DEVICE_ERROR**, **CKR_DEVICE_MEMORY**, **CKR_DEVICE_REMOVED**,
5056 **CKR_FUNCTION_CANCELED**, **CKR_FUNCTION_FAILED**, **CKR_GENERAL_ERROR**,
5057 **CKR_HOST_MEMORY**, **CKR_KEY_FUNCTION_NOT_PERMITTED**, **CKR_KEY_HANDLE_INVALID**,
5058 **CKR_KEY_SIZE_RANGE**, **CKR_KEY_TYPE_INCONSISTENT**, **CKR_MECHANISM_INVALID**,
5059 **CKR_MECHANISM_PARAM_INVALID**, **CKR_OK**, **CKR_OPERATION_ACTIVE**, **CKR_PIN_EXPIRED**,
5060 **CKR_SESSION_CLOSED**, **CKR_SESSION_HANDLE_INVALID**, **CKR_USER_NOT_LOGGED_IN**,
5061 **CKR_OPERATION_CANCEL_FAILED**.

5062 Example: see **C_VerifyRecover**.

5063 5.15.6 C_VerifyRecover

```

5064 CK_DECLARE_FUNCTION(CK_RV, C_VerifyRecover) (
5065     CK_SESSION_HANDLE hSession,
5066     CK_BYTE_PTR pSignature,
5067     CK_ULONG ulSignatureLen,
5068     CK_BYTE_PTR pData,
5069     CK_ULONG_PTR pulDataLen
5070 );

```

5071 **C_VerifyRecover** verifies a signature in a single-part operation, where the data is recovered from the
5072 signature. *hSession* is the session's handle; *pSignature* points to the signature; *ulSignatureLen* is the
5073 length of the signature; *pData* points to the location that receives the recovered data; and *pulDataLen*
5074 points to the location that holds the length of the recovered data.

5075 **C_VerifyRecover** uses the convention described in Section 5.2 on producing output.

5076 The verification operation **MUST** have been initialized with **C_VerifyRecoverInit**. A call to
5077 **C_VerifyRecover** always terminates the active verification operation unless it returns
5078 **CKR_BUFFER_TOO_SMALL** or is a successful call (*i.e.*, one which returns **CKR_OK**) to determine the
5079 length of the buffer needed to hold the recovered data.

5080 A successful call to **C_VerifyRecover** should return either the value CKR_OK (indicating that the
5081 supplied signature is valid) or CKR_SIGNATURE_INVALID (indicating that the supplied signature is
5082 invalid). If the signature can be seen to be invalid purely on the basis of its length, then
5083 CKR_SIGNATURE_LEN_RANGE should be returned. The return codes CKR_SIGNATURE_INVALID
5084 and CKR_SIGNATURE_LEN_RANGE have a higher priority than the return code
5085 CKR_BUFFER_TOO_SMALL, *i.e.*, if **C_VerifyRecover** is supplied with an invalid signature, it will never
5086 return CKR_BUFFER_TOO_SMALL.

5087 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
5088 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE,
5089 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
5090 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
5091 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
5092 CKR_SESSION_HANDLE_INVALID, CKR_SIGNATURE_LEN_RANGE, CKR_SIGNATURE_INVALID,
5093 CKR_TOKEN_RESOURCE_EXCEEDED.

5094 Example:

```
5095 CK_SESSION_HANDLE hSession;  
5096 CK_OBJECT_HANDLE hKey;  
5097 CK_MECHANISM mechanism = {  
5098     CKM_RSA_9796, NULL_PTR, 0  
5099 };  
5100 CK_BYTE data[] = {...};  
5101 CK_ULONG ulDataLen;  
5102 CK_BYTE signature[128];  
5103 CK_RV rv;  
5104  
5105 .  
5106 .  
5107 rv = C_VerifyRecoverInit(hSession, &mechanism, hKey);  
5108 if (rv == CKR_OK) {  
5109     ulDataLen = sizeof(data);  
5110     rv = C_VerifyRecover(  
5111         hSession, signature, sizeof(signature), data, &ulDataLen);  
5112     .  
5113     .  
5114 }
```

5115 5.16 Message-Based Functions for Verifying Signatures and MACs

5116 Message-based verification refers to the process of verifying signatures on multiple messages using the
5117 same verification mechanism and verification key.

5118 Cryptoki provides the following functions for verifying signatures on messages (for the purposes of
5119 Cryptoki, these operations also encompass message authentication codes).

5120 5.16.1 C_MessageVerifyInit

```
5121 CK_DECLARE_FUNCTION(CK_RV, C_MessageVerifyInit) (  
5122     CK_SESSION_HANDLE hSession,  
5123     CK_MECHANISM_PTR pMechanism,
```

```

5124     CK_OBJECT_HANDLE hKey
5125 );

```

5126 **C_MessageVerifyInit** initializes a message-based verification process, preparing a session for one or
5127 more verification operations (where the signature is an appendix to the data) that use the same
5128 verification mechanism and verification key. *hSession* is the session's handle; *pMechanism* points to the
5129 structure that specifies the verification mechanism; *hKey* is the handle of the verification key.

5130 The **CKA_VERIFY** attribute of the verification key, which indicates whether the key supports verification
5131 where the signature is an appendix to the data, MUST be CK_TRUE.

5132 After calling **C_MessageVerifyInit**, the application can either call **C_VerifyMessage** to verify a signature
5133 on a message in a single part; or call **C_VerifyMessageBegin**, followed by **C_VerifyMessageNext** one
5134 or more times, to verify a signature on a message in multiple parts. This may be repeated several times.
5135 The message-based verification process is active until the application calls **C_MessageVerifyFinal** to
5136 finish the message-based verification process.

5137 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
5138 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
5139 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
5140 CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID,
5141 CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,
5142 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,
5143 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

5144 5.16.2 C_VerifyMessage

```

5145 CK_DECLARE_FUNCTION(CK_RV, C_VerifyMessage) (
5146     CK_SESSION_HANDLE hSession,
5147     CK_VOID_PTR pParameter,
5148     CK_ULONG ulParameterLen,
5149     CK_BYTE_PTR pData,
5150     CK_ULONG ulDataLen,
5151     CK_BYTE_PTR pSignature,
5152     CK_ULONG ulSignatureLen
5153 );

```

5154 **C_VerifyMessage** verifies a signature on a message in a single part operation, where the signature is an
5155 appendix to the data. **C_MessageVerifyInit** must previously been called on the session. *hSession* is the
5156 session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the
5157 message verification operation; *pData* points to the data; *ulDataLen* is the length of the data; *pSignature*
5158 points to the signature; *ulSignatureLen* is the length of the signature.

5159 Unlike the *pParameter* parameter of **C_SignMessage**, *pParameter* is always an input parameter.

5160 The message-based verification process MUST have been initialized with **C_MessageVerifyInit**. A call to
5161 **C_VerifyMessage** starts and terminates a message verification operation.

5162 A successful call to **C_VerifyMessage** should return either the value CKR_OK (indicating that the
5163 supplied signature is valid) or CKR_SIGNATURE_INVALID (indicating that the supplied signature is
5164 invalid). If the signature can be seen to be invalid purely on the basis of its length, then
5165 CKR_SIGNATURE_LEN_RANGE should be returned.

5166 **C_VerifyMessage** does not finish the message-based verification process. Additional **C_VerifyMessage**
5167 or **C_VerifyMessageBegin** and **C_VerifyMessageNext** calls may be made on the session.

5168 For most mechanisms, **C_VerifyMessage** is equivalent to **C_VerifyMessageBegin** followed by a
5169 sequence of **C_VerifyMessageNext** operations.

5170 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID,
5171 CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,

5172 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
5173 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
5174 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SIGNATURE_INVALID,
5175 CKR_SIGNATURE_LEN_RANGE, CKR_TOKEN_RESOURCE_EXCEEDED.

5176 5.16.3 C_VerifyMessageBegin

```
5177 CK_DECLARE_FUNCTION(CK_RV, C_VerifyMessageBegin) (  
5178     CK_SESSION_HANDLE hSession,  
5179     CK_VOID_PTR pParameter,  
5180     CK_ULONG ulParameterLen  
5181 );
```

5182 **C_VerifyMessageBegin** begins a multiple-part message verification operation, where the signature is an
5183 appendix to the message. **C_MessageVerifyInit** must previously been called on the session. *hSession* is
5184 the session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for
5185 the message verification operation.

5186 Unlike the *pParameter* parameter of **C_SignMessageBegin**, *pParameter* is always an input parameter.

5187 After calling **C_VerifyMessageBegin**, the application should call **C_VerifyMessageNext** one or more
5188 times to verify a signature on a message in multiple parts. The message verification operation is active
5189 until the application calls **C_VerifyMessageNext** with a non-NULL *pSignature*. To process additional
5190 messages (in single or multiple parts), the application MUST call **C_VerifyMessage** or
5191 **C_VerifyMessageBegin** again.

5192 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
5193 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
5194 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
5195 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,
5196 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

5197 5.16.4 C_VerifyMessageNext

```
5198 CK_DECLARE_FUNCTION(CK_RV, C_VerifyMessageNext) (  
5199     CK_SESSION_HANDLE hSession,  
5200     CK_VOID_PTR pParameter,  
5201     CK_ULONG ulParameterLen,  
5202     CK_BYTE_PTR pDataPart,  
5203     CK_ULONG ulDataPartLen,  
5204     CK_BYTE_PTR pSignature,  
5205     CK_ULONG ulSignatureLen  
5206 );
```

5207 **C_VerifyMessageNext** continues a multiple-part message verification operation, processing another data
5208 part, or finishes a multiple-part message verification operation, checking the signature. *hSession* is the
5209 session's handle, *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the
5210 message verification operation, *pPart* points to the data part; *ulPartLen* is the length of the data part;
5211 *pSignature* points to the signature; *ulSignatureLen* is the length of the signature.

5212 The *pSignature* argument is set to NULL if there is more data part to follow, or set to a non-NULL value
5213 (pointing to the signature to verify) if this is the last data part.

5214 The message verification operation MUST have been started with **C_VerifyMessageBegin**. This function
5215 may be called any number of times in succession. A call to **C_VerifyMessageNext** with a NULL
5216 *pSignature* which results in an error terminates the current message verification operation. A call to

5217 **C_VerifyMessageNext** with a non-NULL *pSignature* always terminates the active message verification
5218 operation.

5219 A successful call to **C_VerifyMessageNext** with a non-NULL *pSignature* should return either the value
5220 CKR_OK (indicating that the supplied signature is valid) or CKR_SIGNATURE_INVALID (indicating that
5221 the supplied signature is invalid). If the signature can be seen to be invalid purely on the basis of its
5222 length, then CKR_SIGNATURE_LEN_RANGE should be returned. In any of these cases, the active
5223 message verifying operation is terminated.

5224 Although the last **C_VerifyMessageNext** call ends the verification of a message, it does not finish the
5225 message-based verification process. Additional **C_VerifyMessage** or **C_VerifyMessageBegin** and
5226 **C_VerifyMessageNext** calls may be made on the session.

5227 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
5228 CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
5229 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
5230 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
5231 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SIGNATURE_INVALID,
5232 CKR_SIGNATURE_LEN_RANGE, CKR_TOKEN_RESOURCE_EXCEEDED.

5233 5.16.5 C_MessageVerifyFinal

```
5234 CK_DECLARE_FUNCTION(CK_RV, C_MessageVerifyFinal) (  
5235     CK_SESSION_HANDLE hSession  
5236 );
```

5237 **C_MessageVerifyFinal** finishes a message-based verification process. *hSession* is the session's handle.

5238 The message-based verification process MUST have been initialized with **C_MessageVerifyInit**.

5239 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
5240 CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
5241 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
5242 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
5243 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
5244 CKR_TOKEN_RESOURCE_EXCEEDED.

5245 5.17 Dual-function cryptographic functions

5246 Cryptoki provides the following functions to perform two cryptographic operations "simultaneously" within
5247 a session. These functions are provided so as to avoid unnecessarily passing data back and forth to and
5248 from a token.

5249 5.17.1 C_DigestEncryptUpdate

```
5250 CK_DECLARE_FUNCTION(CK_RV, C_DigestEncryptUpdate) (  
5251     CK_SESSION_HANDLE hSession,  
5252     CK_BYTE_PTR pPart,  
5253     CK_ULONG ulPartLen,  
5254     CK_BYTE_PTR pEncryptedPart,  
5255     CK_ULONG_PTR pulEncryptedPartLen  
5256 );
```

5257 **C_DigestEncryptUpdate** continues multiple-part digest and encryption operations, processing another
5258 data part. *hSession* is the session's handle; *pPart* points to the data part; *ulPartLen* is the length of the
5259 data part; *pEncryptedPart* points to the location that receives the digested and encrypted data part;
5260 *pulEncryptedPartLen* points to the location that holds the length of the encrypted data part.

5261 **C_DigestEncryptUpdate** uses the convention described in Section 5.2 on producing output. If a
5262 **C_DigestEncryptUpdate** call does not produce encrypted output (because an error occurs, or because

5263 *pEncryptedPart* has the value `NULL_PTR`, or because *pulEncryptedPartLen* is too small to hold the entire
5264 encrypted part output), then no plaintext is passed to the active digest operation.

5265 Digest and encryption operations MUST both be active (they MUST have been initialized with
5266 **C_DigestInit** and **C_EncryptInit**, respectively). This function may be called any number of times in
5267 succession, and may be interspersed with **C_DigestUpdate**, **C_DigestKey**, and **C_EncryptUpdate** calls
5268 (it would be somewhat unusual to intersperse calls to **C_DigestEncryptUpdate** with calls to
5269 **C_DigestKey**, however).

5270 Return values: `CKR_ARGUMENTS_BAD`, `CKR_BUFFER_TOO_SMALL`,
5271 `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DATA_LEN_RANGE`, `CKR_DEVICE_ERROR`,
5272 `CKR_DEVICE_MEMORY`, `CKR_DEVICE_REMOVED`, `CKR_FUNCTION_CANCELED`,
5273 `CKR_FUNCTION_FAILED`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`,
5274 `CKR_OPERATION_NOT_INITIALIZED`, `CKR_SESSION_CLOSED`, `CKR_SESSION_HANDLE_INVALID`.

5275 Example:

```
5276 #define BUF_SZ 512
5277
5278 CK_SESSION_HANDLE hSession;
5279 CK_OBJECT_HANDLE hKey;
5280 CK_BYTE iv[8];
5281 CK_MECHANISM digestMechanism = {
5282     CKM_MD5, NULL_PTR, 0
5283 };
5284 CK_MECHANISM encryptionMechanism = {
5285     CKM_DES_ECB, iv, sizeof(iv)
5286 };
5287 CK_BYTE encryptedData[BUF_SZ];
5288 CK_ULONG ulEncryptedDataLen;
5289 CK_BYTE digest[16];
5290 CK_ULONG ulDigestLen;
5291 CK_BYTE data[(2*BUF_SZ)+8];
5292 CK_RV rv;
5293 int i;
5294
5295 .
5296 .
5297 memset(iv, 0, sizeof(iv));
5298 memset(data, 'A', ((2*BUF_SZ)+5));
5299 rv = C_EncryptInit(hSession, &encryptionMechanism, hKey);
5300 if (rv != CKR_OK) {
5301     .
5302     .
5303 }
5304 rv = C_DigestInit(hSession, &digestMechanism);
5305 if (rv != CKR_OK) {
5306     .
5307     .
```

```

5308 }
5309
5310 ulEncryptedDataLen = sizeof(encryptedData);
5311 rv = C_DigestEncryptUpdate(
5312     hSession,
5313     &data[0], BUF_SZ,
5314     encryptedData, &ulEncryptedDataLen);
5315 .
5316 .
5317 ulEncryptedDataLen = sizeof(encryptedData);
5318 rv = C_DigestEncryptUpdate(
5319     hSession,
5320     &data[BUF_SZ], BUF_SZ,
5321     encryptedData, &ulEncryptedDataLen);
5322 .
5323 .
5324
5325 /*
5326  * The last portion of the buffer needs to be
5327  * handled with separate calls to deal with
5328  * padding issues in ECB mode
5329  */
5330
5331 /* First, complete the digest on the buffer */
5332 rv = C_DigestUpdate(hSession, &data[BUF_SZ*2], 5);
5333 .
5334 .
5335 ulDigestLen = sizeof(digest);
5336 rv = C_DigestFinal(hSession, digest, &ulDigestLen);
5337 .
5338 .
5339
5340 /* Then, pad last part with 3 0x00 bytes, and complete encryption */
5341 for(i=0;i<3;i++)
5342     data[((BUF_SZ*2)+5)+i] = 0x00;
5343
5344 /* Now, get second-to-last piece of ciphertext */
5345 ulEncryptedDataLen = sizeof(encryptedData);
5346 rv = C_EncryptUpdate(
5347     hSession,
5348     &data[BUF_SZ*2], 8,
5349     encryptedData, &ulEncryptedDataLen);
5350 .

```



```

5351 .
5352
5353 /* Get last piece of ciphertext (should have length 0, here) */
5354 ulEncryptedDataLen = sizeof(encryptedData);
5355 rv = C_EncryptFinal(hSession, encryptedData, &ulEncryptedDataLen);
5356 .
5357 .

```

5358 5.17.2 C_DecryptDigestUpdate

```

5359 CK_DECLARE_FUNCTION(CK_RV, C_DecryptDigestUpdate) (
5360     CK_SESSION_HANDLE hSession,
5361     CK_BYTE_PTR pEncryptedPart,
5362     CK_ULONG ulEncryptedPartLen,
5363     CK_BYTE_PTR pPart,
5364     CK_ULONG_PTR pulPartLen
5365 );

```

5366 **C_DecryptDigestUpdate** continues a multiple-part combined decryption and digest operation,
5367 processing another data part. *hSession* is the session's handle; *pEncryptedPart* points to the encrypted
5368 data part; *ulEncryptedPartLen* is the length of the encrypted data part; *pPart* points to the location that
5369 receives the recovered data part; *pulPartLen* points to the location that holds the length of the recovered
5370 data part.

5371 **C_DecryptDigestUpdate** uses the convention described in Section 5.2 on producing output. If a
5372 **C_DecryptDigestUpdate** call does not produce decrypted output (because an error occurs, or because
5373 *pPart* has the value `NULL_PTR`, or because *pulPartLen* is too small to hold the entire decrypted part
5374 output), then no plaintext is passed to the active digest operation.

5375 Decryption and digesting operations **MUST** both be active (they **MUST** have been initialized with
5376 **C_DecryptInit** and **C_DigestInit**, respectively). This function may be called any number of times in
5377 succession, and may be interspersed with **C_DecryptUpdate**, **C_DigestUpdate**, and **C_DigestKey** calls
5378 (it would be somewhat unusual to intersperse calls to **C_DigestEncryptUpdate** with calls to
5379 **C_DigestKey**, however).

5380 Use of **C_DecryptDigestUpdate** involves a pipelining issue that does not arise when using
5381 **C_DigestEncryptUpdate**, the "inverse function" of **C_DecryptDigestUpdate**. This is because when
5382 **C_DigestEncryptUpdate** is called, precisely the same input is passed to both the active digesting
5383 operation and the active encryption operation; however, when **C_DecryptDigestUpdate** is called, the
5384 input passed to the active digesting operation is the *output* of the active decryption operation. This issue
5385 comes up only when the mechanism used for decryption performs padding.

5386 In particular, envision a 24-byte ciphertext which was obtained by encrypting an 18-byte plaintext with
5387 DES in CBC mode with PKCS padding. Consider an application which will simultaneously decrypt this
5388 ciphertext and digest the original plaintext thereby obtained.

5389 After initializing decryption and digesting operations, the application passes the 24-byte ciphertext (3 DES
5390 blocks) into **C_DecryptDigestUpdate**. **C_DecryptDigestUpdate** returns exactly 16 bytes of plaintext,
5391 since at this point, Cryptoki doesn't know if there's more ciphertext coming, or if the last block of
5392 ciphertext held any padding. These 16 bytes of plaintext are passed into the active digesting operation.

5393 Since there is no more ciphertext, the application calls **C_DecryptFinal**. This tells Cryptoki that there's
5394 no more ciphertext coming, and the call returns the last 2 bytes of plaintext. However, since the active
5395 decryption and digesting operations are linked *only* through the **C_DecryptDigestUpdate** call, these 2
5396 bytes of plaintext are *not* passed on to be digested.

5397 A call to **C_DigestFinal**, therefore, would compute the message digest of *the first 16 bytes of the*
5398 *plaintext*, not the message digest of the entire plaintext. It is crucial that, before **C_DigestFinal** is called,
5399 the last 2 bytes of plaintext get passed into the active digesting operation via a **C_DigestUpdate** call.

5400 Because of this, it is critical that when an application uses a padded decryption mechanism with
5401 **C_DecryptDigestUpdate**, it knows exactly how much plaintext has been passed into the active digesting
5402 operation. *Extreme caution is warranted when using a padded decryption mechanism with*
5403 **C_DecryptDigestUpdate**.

5404 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
5405 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
5406 CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID,
5407 CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
5408 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
5409 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

5410 Example:

```
5411 #define BUF_SZ 512
5412
5413 CK_SESSION_HANDLE hSession;
5414 CK_OBJECT_HANDLE hKey;
5415 CK_BYTE iv[8];
5416 CK_MECHANISM decryptionMechanism = {
5417     CKM_DES_ECB, iv, sizeof(iv)
5418 };
5419 CK_MECHANISM digestMechanism = {
5420     CKM_MD5, NULL_PTR, 0
5421 };
5422 CK_BYTE encryptedData[(2*BUF_SZ)+8];
5423 CK_BYTE digest[16];
5424 CK_ULONG ulDigestLen;
5425 CK_BYTE data[BUF_SZ];
5426 CK_ULONG ulDataLen, ulLastUpdateSize;
5427 CK_RV rv;
5428
5429 .
5430 .
5431 memset(iv, 0, sizeof(iv));
5432 memset(encryptedData, 'A', ((2*BUF_SZ)+8));
5433 rv = C_DecryptInit(hSession, &decryptionMechanism, hKey);
5434 if (rv != CKR_OK) {
5435     .
5436     .
5437 }
5438 rv = C_DigestInit(hSession, &digestMechanism);
5439 if (rv != CKR_OK){
5440     .
5441     .
5442 }
5443
5444 ulDataLen = sizeof(data);
```

```

5445 rv = C_DecryptDigestUpdate(
5446     hSession,
5447     &encryptedData[0], BUF_SZ,
5448     data, &ulDataLen);
5449 .
5450 .
5451 ulDataLen = sizeof(data);
5452 rv = C_DecryptDigestUpdate(
5453     hSession,
5454     &encryptedData[BUF_SZ], BUF_SZ,
5455     data, &ulDataLen);
5456 .
5457 .
5458
5459 /*
5460  * The last portion of the buffer needs to be handled with
5461  * separate calls to deal with padding issues in ECB mode
5462  */
5463
5464 /* First, complete the decryption of the buffer */
5465 ulLastUpdateSize = sizeof(data);
5466 rv = C_DecryptUpdate(
5467     hSession,
5468     &encryptedData[BUF_SZ*2], 8,
5469     data, &ulLastUpdateSize);
5470 .
5471 .
5472 /* Get last piece of plaintext (should have length 0, here) */
5473 ulDataLen = sizeof(data)-ulLastUpdateSize;
5474 rv = C_DecryptFinal(hSession, &data[ulLastUpdateSize], &ulDataLen);
5475 if (rv != CKR_OK) {
5476     .
5477     .
5478 }
5479
5480 /* Digest last bit of plaintext */
5481 rv = C_DigestUpdate(hSession, &data[BUF_SZ*2], 5);
5482 if (rv != CKR_OK) {
5483     .
5484     .
5485 }
5486 ulDigestLen = sizeof(digest);
5487 rv = C_DigestFinal(hSession, digest, &ulDigestLen);

```

```

5488 if (rv != CKR_OK) {
5489     .
5490     .
5491 }

```

5492 5.17.3 C_SignEncryptUpdate

```

5493 CK_DECLARE_FUNCTION(CK_RV, C_SignEncryptUpdate) (
5494     CK_SESSION_HANDLE hSession,
5495     CK_BYTE_PTR pPart,
5496     CK_ULONG ulPartLen,
5497     CK_BYTE_PTR pEncryptedPart,
5498     CK_ULONG_PTR pulEncryptedPartLen
5499 );

```

5500 **C_SignEncryptUpdate** continues a multiple-part combined signature and encryption operation,
5501 processing another data part. *hSession* is the session's handle; *pPart* points to the data part; *ulPartLen* is
5502 the length of the data part; *pEncryptedPart* points to the location that receives the digested and encrypted
5503 data part; and *pulEncryptedPartLen* points to the location that holds the length of the encrypted data part.

5504 **C_SignEncryptUpdate** uses the convention described in Section 5.2 on producing output. If a
5505 **C_SignEncryptUpdate** call does not produce encrypted output (because an error occurs, or because
5506 *pEncryptedPart* has the value `NULL_PTR`, or because *pulEncryptedPartLen* is too small to hold the entire
5507 encrypted part output), then no plaintext is passed to the active signing operation.

5508 Signature and encryption operations **MUST** both be active (they **MUST** have been initialized with
5509 **C_SignInit** and **C_EncryptInit**, respectively). This function may be called any number of times in
5510 succession, and may be interspersed with **C_SignUpdate** and **C_EncryptUpdate** calls.

5511 Return values: `CKR_ARGUMENTS_BAD`, `CKR_BUFFER_TOO_SMALL`,
5512 `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DATA_LEN_RANGE`, `CKR_DEVICE_ERROR`,
5513 `CKR_DEVICE_MEMORY`, `CKR_DEVICE_REMOVED`, `CKR_FUNCTION_CANCELED`,
5514 `CKR_FUNCTION_FAILED`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`,
5515 `CKR_OPERATION_NOT_INITIALIZED`, `CKR_SESSION_CLOSED`, `CKR_SESSION_HANDLE_INVALID`,
5516 `CKR_USER_NOT_LOGGED_IN`.

5517 Example:

```

5518 #define BUF_SZ 512
5519
5520 CK_SESSION_HANDLE hSession;
5521 CK_OBJECT_HANDLE hEncryptionKey, hMacKey;
5522 CK_BYTE iv[8];
5523 CK_MECHANISM signMechanism = {
5524     CKM_DES_MAC, NULL_PTR, 0
5525 };
5526 CK_MECHANISM encryptionMechanism = {
5527     CKM_DES_ECB, iv, sizeof(iv)
5528 };
5529 CK_BYTE encryptedData[BUF_SZ];
5530 CK_ULONG ulEncryptedDataLen;
5531 CK_BYTE MAC[4];
5532 CK_ULONG ulMacLen;
5533 CK_BYTE data[(2*BUF_SZ)+8];

```

```

5534 CK_RV rv;
5535 int i;
5536
5537 .
5538 .
5539 memset(iv, 0, sizeof(iv));
5540 memset(data, 'A', ((2*BUF_SZ)+5));
5541 rv = C_EncryptInit(hSession, &encryptionMechanism, hEncryptionKey);
5542 if (rv != CKR_OK) {
5543     .
5544     .
5545 }
5546 rv = C_SignInit(hSession, &signMechanism, hMacKey);
5547 if (rv != CKR_OK) {
5548     .
5549     .
5550 }
5551
5552 ulEncryptedDataLen = sizeof(encryptedData);
5553 rv = C_SignEncryptUpdate(
5554     hSession,
5555     &data[0], BUF_SZ,
5556     encryptedData, &ulEncryptedDataLen);
5557 .
5558 .
5559 ulEncryptedDataLen = sizeof(encryptedData);
5560 rv = C_SignEncryptUpdate(
5561     hSession,
5562     &data[BUF_SZ], BUF_SZ,
5563     encryptedData, &ulEncryptedDataLen);
5564 .
5565 .
5566
5567 /*
5568  * The last portion of the buffer needs to be handled with
5569  * separate calls to deal with padding issues in ECB mode
5570  */
5571
5572 /* First, complete the signature on the buffer */
5573 rv = C_SignUpdate(hSession, &data[BUF_SZ*2], 5);
5574 .
5575 .
5576 ulMacLen = sizeof(MAC);

```

```

5577 rv = C_SignFinal(hSession, MAC, &ulMacLen);
5578 .
5579 .
5580
5581 /* Then pad last part with 3 0x00 bytes, and complete encryption */
5582 for(i=0;i<3;i++)
5583     data[((BUF_SZ*2)+5)+i] = 0x00;
5584
5585 /* Now, get second-to-last piece of ciphertext */
5586 ulEncryptedDataLen = sizeof(encryptedData);
5587 rv = C_EncryptUpdate(
5588     hSession,
5589     &data[BUF_SZ*2], 8,
5590     encryptedData, &ulEncryptedDataLen);
5591 .
5592 .
5593
5594 /* Get last piece of ciphertext (should have length 0, here) */
5595 ulEncryptedDataLen = sizeof(encryptedData);
5596 rv = C_EncryptFinal(hSession, encryptedData, &ulEncryptedDataLen);
5597 .
5598 .

```

5599 5.17.4 C_DecryptVerifyUpdate

```

5600 CK_DECLARE_FUNCTION(CK_RV, C_DecryptVerifyUpdate)(
5601     CK_SESSION_HANDLE hSession,
5602     CK_BYTE_PTR pEncryptedPart,
5603     CK_ULONG ulEncryptedPartLen,
5604     CK_BYTE_PTR pPart,
5605     CK_ULONG_PTR pulPartLen
5606 );

```

5607 **C_DecryptVerifyUpdate** continues a multiple-part combined decryption and verification operation,
5608 processing another data part. *hSession* is the session's handle; *pEncryptedPart* points to the encrypted
5609 data; *ulEncryptedPartLen* is the length of the encrypted data; *pPart* points to the location that receives the
5610 recovered data; and *pulPartLen* points to the location that holds the length of the recovered data.

5611 **C_DecryptVerifyUpdate** uses the convention described in Section 5.2 on producing output. If a
5612 **C_DecryptVerifyUpdate** call does not produce decrypted output (because an error occurs, or because
5613 *pPart* has the value `NULL_PTR`, or because *pulPartLen* is too small to hold the entire encrypted part
5614 output), then no plaintext is passed to the active verification operation.

5615 Decryption and signature operations **MUST** both be active (they **MUST** have been initialized with
5616 **C_DecryptInit** and **C_VerifyInit**, respectively). This function may be called any number of times in
5617 succession, and may be interspersed with **C_DecryptUpdate** and **C_VerifyUpdate** calls.

5618 Use of **C_DecryptVerifyUpdate** involves a pipelining issue that does not arise when using
5619 **C_SignEncryptUpdate**, the "inverse function" of **C_DecryptVerifyUpdate**. This is because when
5620 **C_SignEncryptUpdate** is called, precisely the same input is passed to both the active signing operation
5621 and the active encryption operation; however, when **C_DecryptVerifyUpdate** is called, the input passed

to the active verifying operation is the *output* of the active decryption operation. This issue comes up only when the mechanism used for decryption performs padding.

In particular, envision a 24-byte ciphertext which was obtained by encrypting an 18-byte plaintext with DES in CBC mode with PKCS padding. Consider an application which will simultaneously decrypt this ciphertext and verify a signature on the original plaintext thereby obtained.

After initializing decryption and verification operations, the application passes the 24-byte ciphertext (3 DES blocks) into **C_DecryptVerifyUpdate**. **C_DecryptVerifyUpdate** returns exactly 16 bytes of plaintext, since at this point, Cryptoki doesn't know if there's more ciphertext coming, or if the last block of ciphertext held any padding. These 16 bytes of plaintext are passed into the active verification operation.

Since there is no more ciphertext, the application calls **C_DecryptFinal**. This tells Cryptoki that there's no more ciphertext coming, and the call returns the last 2 bytes of plaintext. However, since the active decryption and verification operations are linked *only* through the **C_DecryptVerifyUpdate** call, these 2 bytes of plaintext are *not* passed on to the verification mechanism.

A call to **C_VerifyFinal**, therefore, would verify whether or not the signature supplied is a valid signature on the *first 16 bytes of the plaintext*, not on the entire plaintext. It is crucial that, before **C_VerifyFinal** is called, the last 2 bytes of plaintext get passed into the active verification operation via a **C_VerifyUpdate** call.

Because of this, it is critical that when an application uses a padded decryption mechanism with **C_DecryptVerifyUpdate**, it knows exactly how much plaintext has been passed into the active verification operation. *Extreme caution is warranted when using a padded decryption mechanism with C_DecryptVerifyUpdate.*

Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID, CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example:

```
#define BUF_SZ 512

CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hDecryptionKey, hMacKey;
CK_BYTE iv[8];
CK_MECHANISM decryptionMechanism = {
    CKM_DES_ECB, iv, sizeof(iv)
};
CK_MECHANISM verifyMechanism = {
    CKM_DES_MAC, NULL_PTR, 0
};
CK_BYTE encryptedData[(2*BUF_SZ)+8];
CK_BYTE MAC[4];
CK_ULONG ulMacLen;
CK_BYTE data[BUF_SZ];
CK_ULONG ulDataLen, ulLastUpdateSize;
CK_RV rv;

.
```

```

5670 memset(iv, 0, sizeof(iv));
5671 memset(encryptedData, 'A', ((2*BUF_SZ)+8));
5672 rv = C_DecryptInit(hSession, &decryptionMechanism, hDecryptionKey);
5673 if (rv != CKR_OK) {
5674     .
5675     .
5676 }
5677 rv = C_VerifyInit(hSession, &verifyMechanism, hMacKey);
5678 if (rv != CKR_OK){
5679     .
5680     .
5681 }
5682
5683 ulDataLen = sizeof(data);
5684 rv = C_DecryptVerifyUpdate(
5685     hSession,
5686     &encryptedData[0], BUF_SZ,
5687     data, &ulDataLen);
5688 .
5689 .
5690 ulDataLen = sizeof(data);
5691 rv = C_DecryptVerifyUpdate(
5692     hSession,
5693     &encryptedData[BUF_SZ], BUF_SZ,
5694     data, &uldataLen);
5695 .
5696 .
5697
5698 /*
5699  * The last portion of the buffer needs to be handled with
5700  * separate calls to deal with padding issues in ECB mode
5701  */
5702
5703 /* First, complete the decryption of the buffer */
5704 ulLastUpdateSize = sizeof(data);
5705 rv = C_DecryptUpdate(
5706     hSession,
5707     &encryptedData[BUF_SZ*2], 8,
5708     data, &ulLastUpdateSize);
5709 .
5710 .
5711 /* Get last little piece of plaintext. Should have length 0 */
5712 ulDataLen = sizeof(data)-ulLastUpdateSize;

```

```

5713 rv = C_DecryptFinal(hSession, &data[ulLastUpdateSize], &ulDataLen);
5714 if (rv != CKR_OK) {
5715     .
5716     .
5717 }
5718
5719 /* Send last bit of plaintext to verification operation */
5720 rv = C_VerifyUpdate(hSession, &data[BUF_SZ*2], 5);
5721 if (rv != CKR_OK) {
5722     .
5723     .
5724 }
5725 rv = C_VerifyFinal(hSession, MAC, ulMacLen);
5726 if (rv == CKR_SIGNATURE_INVALID) {
5727     .
5728     .
5729 }

```

5730 5.18 Key management functions

5731 Cryptoki provides the following functions for key management:

5732 5.18.1 C_GenerateKey

```

5733 CK_DECLARE_FUNCTION(CK_RV, C_GenerateKey) (
5734     CK_SESSION_HANDLE hSession
5735     CK_MECHANISM_PTR pMechanism,
5736     CK_ATTRIBUTE_PTR pTemplate,
5737     CK_ULONG ulCount,
5738     CK_OBJECT_HANDLE_PTR phKey
5739 );

```

5740 **C_GenerateKey** generates a secret key or set of domain parameters, creating a new object. *hSession* is
5741 the session's handle; *pMechanism* points to the generation mechanism; *pTemplate* points to the template
5742 for the new key or set of domain parameters; *ulCount* is the number of attributes in the template; *phKey*
5743 points to the location that receives the handle of the new key or set of domain parameters.

5744 If the generation mechanism is for domain parameter generation, the **CKA_CLASS** attribute will have the
5745 value CKO_DOMAIN_PARAMETERS; otherwise, it will have the value CKO_SECRET_KEY.

5746 Since the type of key or domain parameters to be generated is implicit in the generation mechanism, the
5747 template does not need to supply a key type. If it does supply a key type which is inconsistent with the
5748 generation mechanism, **C_GenerateKey** fails and returns the error code
5749 CKR_TEMPLATE_INCONSISTENT. The CKA_CLASS attribute is treated similarly.

5750 If a call to **C_GenerateKey** cannot support the precise template supplied to it, it will fail and return without
5751 creating an object.

5752 The object created by a successful call to **C_GenerateKey** will have its **CKA_LOCAL** attribute set to
5753 CK_TRUE. In addition, the object created will have a value for CKA_UNIQUE_ID generated and
5754 assigned (See Section 4.4.1).

5755 Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY,
5756 CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID,
5757 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_CURVE_NOT_SUPPORTED, CKR_DEVICE_ERROR,

5758 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED,
 5759 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY,
 5760 CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK,
 5761 CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED,
 5762 CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCOMPLETE,
 5763 CKR_TEMPLATE_INCONSISTENT, CKR_TOKEN_WRITE_PROTECTED,
 5764 CKR_USER_NOT_LOGGED_IN.

5765 Example:

```
5766 CK_SESSION_HANDLE hSession;
5767 CK_OBJECT_HANDLE hKey;
5768 CK_MECHANISM mechanism = {
5769     CKM_DES_KEY_GEN, NULL_PTR, 0
5770 };
5771 CK_RV rv;
5772
5773 .
5774 .
5775 rv = C_GenerateKey(hSession, &mechanism, NULL_PTR, 0, &hKey);
5776 if (rv == CKR_OK) {
5777     .
5778     .
5779 }
```

5780 5.18.2 C_GenerateKeyPair

```
5781 CK_DECLARE_FUNCTION(CK_RV, C_GenerateKeyPair) (
5782     CK_SESSION_HANDLE hSession,
5783     CK_MECHANISM_PTR pMechanism,
5784     CK_ATTRIBUTE_PTR pPublicKeyTemplate,
5785     CK_ULONG ulPublicKeyAttributeCount,
5786     CK_ATTRIBUTE_PTR pPrivateKeyTemplate,
5787     CK_ULONG ulPrivateKeyAttributeCount,
5788     CK_OBJECT_HANDLE_PTR phPublicKey,
5789     CK_OBJECT_HANDLE_PTR phPrivateKey
5790 );
```

5791 **C_GenerateKeyPair** generates a public/private key pair, creating new key objects. *hSession* is the
 5792 session's handle; *pMechanism* points to the key generation mechanism; *pPublicKeyTemplate* points to
 5793 the template for the public key; *ulPublicKeyAttributeCount* is the number of attributes in the public-key
 5794 template; *pPrivateKeyTemplate* points to the template for the private key; *ulPrivateKeyAttributeCount* is
 5795 the number of attributes in the private-key template; *phPublicKey* points to the location that receives the
 5796 handle of the new public key; *phPrivateKey* points to the location that receives the handle of the new
 5797 private key.

5798 Since the types of keys to be generated are implicit in the key pair generation mechanism, the templates
 5799 do not need to supply key types. If one of the templates does supply a key type which is inconsistent with
 5800 the key generation mechanism, **C_GenerateKeyPair** fails and returns the error code
 5801 CKR_TEMPLATE_INCONSISTENT. The CKA_CLASS attribute is treated similarly.

5802 If a call to **C_GenerateKeyPair** cannot support the precise templates supplied to it, it will fail and return
 5803 without creating any key objects.

5804 A call to **C_GenerateKeyPair** will never create just one key and return. A call can fail, and create no
 5805 keys; or it can succeed, and create a matching public/private key pair.

5806 The key objects created by a successful call to **C_GenerateKeyPair** will have their **CKA_LOCAL**
5807 attributes set to CK_TRUE. In addition, the key objects created will both have values for
5808 CKA_UNIQUE_ID generated and assigned (See Section 4.4.1).

5809 *Note carefully the order of the arguments to **C_GenerateKeyPair**. The last two arguments do not have*
5810 *the same order as they did in the original Cryptoki Version 1.0 document. The order of these two*
5811 *arguments has caused some unfortunate confusion.*

5812 Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY,
5813 CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID,
5814 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_CURVE_NOT_SUPPORTED, CKR_DEVICE_ERROR,
5815 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DOMAIN_PARAMS_INVALID,
5816 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
5817 CKR_HOST_MEMORY, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID,
5818 CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED,
5819 CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCOMPLETE,
5820 CKR_TEMPLATE_INCONSISTENT, CKR_TOKEN_WRITE_PROTECTED,
5821 CKR_USER_NOT_LOGGED_IN.

5822 Example:

```
5823 CK_SESSION_HANDLE hSession;  
5824 CK_OBJECT_HANDLE hPublicKey, hPrivateKey;  
5825 CK_MECHANISM mechanism = {  
5826     CKM_RSA_PKCS_KEY_PAIR_GEN, NULL_PTR, 0  
5827 };  
5828 CK_ULONG modulusBits = 768;  
5829 CK_BYTE publicExponent[] = { 3 };  
5830 CK_BYTE subject[] = {...};  
5831 CK_BYTE id[] = {123};  
5832 CK_BBOOL true = CK_TRUE;  
5833 CK_ATTRIBUTE publicKeyTemplate[] = {  
5834     {CKA_ENCRYPT, &true, sizeof(true)},  
5835     {CKA_VERIFY, &true, sizeof(true)},  
5836     {CKA_WRAP, &true, sizeof(true)},  
5837     {CKA_MODULUS_BITS, &modulusBits, sizeof(modulusBits)},  
5838     {CKA_PUBLIC_EXPONENT, publicExponent, sizeof (publicExponent)}  
5839 };  
5840 CK_ATTRIBUTE privateKeyTemplate[] = {  
5841     {CKA_TOKEN, &true, sizeof(true)},  
5842     {CKA_PRIVATE, &true, sizeof(true)},  
5843     {CKA_SUBJECT, subject, sizeof(subject)},  
5844     {CKA_ID, id, sizeof(id)},  
5845     {CKA_SENSITIVE, &true, sizeof(true)},  
5846     {CKA_DECRYPT, &true, sizeof(true)},  
5847     {CKA_SIGN, &true, sizeof(true)},  
5848     {CKA_UNWRAP, &true, sizeof(true)}  
5849 };  
5850 CK_RV rv;  
5851
```

```

5852 rv = C_GenerateKeyPair(
5853     hSession, &mechanism,
5854     publicKeyTemplate, 5,
5855     privateKeyTemplate, 8,
5856     &hPublicKey, &hPrivateKey);
5857 if (rv == CKR_OK) {
5858     .
5859     .
5860 }

```

5861 5.18.3 C_WrapKey

```

5862 CK_DECLARE_FUNCTION(CK_RV, C_WrapKey) (
5863     CK_SESSION_HANDLE hSession,
5864     CK_MECHANISM_PTR pMechanism,
5865     CK_OBJECT_HANDLE hWrappingKey,
5866     CK_OBJECT_HANDLE hKey,
5867     CK_BYTE_PTR pWrappedKey,
5868     CK_ULONG_PTR pulWrappedKeyLen
5869 );

```

5870 **C_WrapKey** wraps (i.e., encrypts) a private or secret key. *hSession* is the session's handle; *pMechanism*
5871 points to the wrapping mechanism; *hWrappingKey* is the handle of the wrapping key; *hKey* is the handle
5872 of the key to be wrapped; *pWrappedKey* points to the location that receives the wrapped key; and
5873 *pulWrappedKeyLen* points to the location that receives the length of the wrapped key.

5874 **C_WrapKey** uses the convention described in Section 5.2 on producing output.

5875 The **CKA_WRAP** attribute of the wrapping key, which indicates whether the key supports wrapping,
5876 MUST be CK_TRUE. The **CKA_EXTRACTABLE** attribute of the key to be wrapped MUST also be
5877 CK_TRUE.

5878 If the key to be wrapped cannot be wrapped for some token-specific reason, despite its having its
5879 **CKA_EXTRACTABLE** attribute set to CK_TRUE, then **C_WrapKey** fails with error code
5880 CKR_KEY_NOT_WRAPPABLE. If it cannot be wrapped with the specified wrapping key and mechanism
5881 solely because of its length, then **C_WrapKey** fails with error code CKR_KEY_SIZE_RANGE.

5882 **C_WrapKey** can be used in the following situations:

- 5883 • To wrap any secret key with a public key that supports encryption and decryption.
- 5884 • To wrap any secret key with any other secret key. Consideration MUST be given to key size and
5885 mechanism strength or the token may not allow the operation.
- 5886 • To wrap a private key with any secret key.

5887 Of course, tokens vary in which types of keys can actually be wrapped with which mechanisms.

5888 To partition the wrapping keys so they can only wrap a subset of extractable keys the attribute
5889 CKA_WRAP_TEMPLATE can be used on the wrapping key to specify an attribute set that will be
5890 compared against the attributes of the key to be wrapped. If all attributes match according to the
5891 C_FindObject rules of attribute matching then the wrap will proceed. The value of this attribute is an
5892 attribute template and the size is the number of items in the template times the size of CK_ATTRIBUTE. If
5893 this attribute is not supplied then any template is acceptable. If an attribute is not present, it will not be
5894 checked. If any attribute mismatch occurs on an attempt to wrap a key then the function SHALL return
5895 CKR_KEY_HANDLE_INVALID.

5896 Return Values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
5897 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
5898 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
5899 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_HANDLE_INVALID,

5900 CKR_KEY_NOT_WRAPPABLE, CKR_KEY_SIZE_RANGE, CKR_KEY_UNEXTRACTABLE,
 5901 CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK,
 5902 CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED,
 5903 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
 5904 CKR_WRAPPING_KEY_HANDLE_INVALID, CKR_WRAPPING_KEY_SIZE_RANGE,
 5905 CKR_WRAPPING_KEY_TYPE_INCONSISTENT.

5906 Example:

```
5907 CK_SESSION_HANDLE hSession;
5908 CK_OBJECT_HANDLE hWrappingKey, hKey;
5909 CK_MECHANISM mechanism = {
5910     CKM_DES3_ECB, NULL_PTR, 0
5911 };
5912 CK_BYTE wrappedKey[8];
5913 CK_ULONG ulWrappedKeyLen;
5914 CK_RV rv;
5915
5916 .
5917 .
5918 ulWrappedKeyLen = sizeof(wrappedKey);
5919 rv = C_WrapKey(
5920     hSession, &mechanism,
5921     hWrappingKey, hKey,
5922     wrappedKey, &ulWrappedKeyLen);
5923 if (rv == CKR_OK) {
5924     .
5925     .
5926 }
```

5927 5.18.4 C_UnwrapKey

```
5928 CK_DECLARE_FUNCTION(CK_RV, C_UnwrapKey) (
5929     CK_SESSION_HANDLE hSession,
5930     CK_MECHANISM_PTR pMechanism,
5931     CK_OBJECT_HANDLE hUnwrappingKey,
5932     CK_BYTE_PTR pWrappedKey,
5933     CK_ULONG ulWrappedKeyLen,
5934     CK_ATTRIBUTE_PTR pTemplate,
5935     CK_ULONG ulAttributeCount,
5936     CK_OBJECT_HANDLE_PTR phKey
5937 );
```

5938 **C_UnwrapKey** unwraps (i.e. decrypts) a wrapped key, creating a new private key or secret key object.
 5939 *hSession* is the session's handle; *pMechanism* points to the unwrapping mechanism; *hUnwrappingKey* is
 5940 the handle of the unwrapping key; *pWrappedKey* points to the wrapped key; *ulWrappedKeyLen* is the
 5941 length of the wrapped key; *pTemplate* points to the template for the new key; *ulAttributeCount* is the
 5942 number of attributes in the template; *phKey* points to the location that receives the handle of the
 5943 recovered key.

5944 The **CKA_UNWRAP** attribute of the unwrapping key, which indicates whether the key supports
 5945 unwrapping, MUST be CK_TRUE.

5946 The new key will have the **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, and the
5947 **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE. The **CKA_EXTRACTABLE** attribute is by
5948 default set to CK_TRUE.

5949 Some mechanisms may modify, or attempt to modify, the contents of the pMechanism structure at the
5950 same time that the key is unwrapped.

5951 If a call to **C_UnwrapKey** cannot support the precise template supplied to it, it will fail and return without
5952 creating any key object.

5953 The key object created by a successful call to **C_UnwrapKey** will have its **CKA_LOCAL** attribute set to
5954 CK_FALSE. In addition, the object created will have a value for CKA_UNIQUE_ID generated and
5955 assigned (See Section 4.4.1).

5956 To partition the unwrapping keys so they can only unwrap a subset of keys the attribute
5957 CKA_UNWRAP_TEMPLATE can be used on the unwrapping key to specify an attribute set that will be
5958 added to attributes of the key to be unwrapped. If the attributes do not conflict with the user supplied
5959 attribute template, in 'pTemplate', then the unwrap will proceed. The value of this attribute is an attribute
5960 template and the size is the number of items in the template times the size of CK_ATTRIBUTE. If this
5961 attribute is not present on the unwrapping key then no additional attributes will be added. If any attribute
5962 conflict occurs on an attempt to unwrap a key then the function SHALL return
5963 CKR_TEMPLATE_INCONSISTENT.

5964 Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY,
5965 CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID,
5966 CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED,
5967 CKR_CURVE_NOT_SUPPORTED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
5968 CKR_DEVICE_REMOVED, CKR_DOMAIN_PARAMS_INVALID, CKR_FUNCTION_CANCELED,
5969 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY,
5970 CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK,
5971 CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED,
5972 CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCOMPLETE,
5973 CKR_TEMPLATE_INCONSISTENT, CKR_TOKEN_WRITE_PROTECTED,
5974 CKR_UNWRAPPING_KEY_HANDLE_INVALID, CKR_UNWRAPPING_KEY_SIZE_RANGE,
5975 CKR_UNWRAPPING_KEY_TYPE_INCONSISTENT, CKR_USER_NOT_LOGGED_IN,
5976 CKR_WRAPPED_KEY_INVALID, CKR_WRAPPED_KEY_LEN_RANGE.

5977 Example:

```
5978 CK_SESSION_HANDLE hSession;
5979 CK_OBJECT_HANDLE hUnwrappingKey, hKey;
5980 CK_MECHANISM mechanism = {
5981     CKM_DES3_ECB, NULL_PTR, 0
5982 };
5983 CK_BYTE wrappedKey[8] = {...};
5984 CK_OBJECT_CLASS keyClass = CKO_SECRET_KEY;
5985 CK_KEY_TYPE keyType = CKK_DES;
5986 CK_BBOOL true = CK_TRUE;
5987 CK_ATTRIBUTE template[] = {
5988     {CKA_CLASS, &keyClass, sizeof(keyClass)},
5989     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
5990     {CKA_ENCRYPT, &true, sizeof(true)},
5991     {CKA_DECRYPT, &true, sizeof(true)}
5992 };
5993 CK_RV rv;
5994
```

```

5995 .
5996 .
5997 rv = C_UnwrapKey(
5998     hSession, &mechanism, hUnwrappingKey,
5999     wrappedKey, sizeof(wrappedKey), template, 4, &hKey);
6000 if (rv == CKR_OK) {
6001     .
6002     .
6003 }

```

6004 5.18.5 C_DeriveKey

```

6005 CK_DECLARE_FUNCTION(CK_RV, C_DeriveKey) (
6006     CK_SESSION_HANDLE hSession,
6007     CK_MECHANISM_PTR pMechanism,
6008     CK_OBJECT_HANDLE hBaseKey,
6009     CK_ATTRIBUTE_PTR pTemplate,
6010     CK_ULONG ulAttributeCount,
6011     CK_OBJECT_HANDLE_PTR phKey
6012 );

```

6013 **C_DeriveKey** derives a key from a base key, creating a new key object. *hSession* is the session's
6014 handle; *pMechanism* points to a structure that specifies the key derivation mechanism; *hBaseKey* is the
6015 handle of the base key; *pTemplate* points to the template for the new key; *ulAttributeCount* is the number
6016 of attributes in the template; and *phKey* points to the location that receives the handle of the derived key.

6017 The values of the **CKA_SENSITIVE**, **CKA_ALWAYS_SENSITIVE**, **CKA_EXTRACTABLE**, and
6018 **CKA_NEVER_EXTRACTABLE** attributes for the base key affect the values that these attributes can hold
6019 for the newly-derived key. See the description of each particular key-derivation mechanism in Section
6020 5.21.2 for any constraints of this type.

6021 If a call to **C_DeriveKey** cannot support the precise template supplied to it, it will fail and return without
6022 creating any key object.

6023 The key object created by a successful call to **C_DeriveKey** will have its **CKA_LOCAL** attribute set to
6024 CK_FALSE. In addition, the object created will have a value for CKA_UNIQUE_ID generated and
6025 assigned (See Section 4.4.1).

6026 Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY,
6027 CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID,
6028 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_CURVE_NOT_SUPPORTED, CKR_DEVICE_ERROR,
6029 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DOMAIN_PARAMS_INVALID,
6030 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
6031 CKR_HOST_MEMORY, CKR_KEY_HANDLE_INVALID, CKR_KEY_SIZE_RANGE,
6032 CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,
6033 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,
6034 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY,
6035 CKR_TEMPLATE_INCOMPLETE, CKR_TEMPLATE_INCONSISTENT,
6036 CKR_TOKEN_WRITE_PROTECTED, CKR_USER_NOT_LOGGED_IN.

6037 Example:

```

6038 CK_SESSION_HANDLE hSession;
6039 CK_OBJECT_HANDLE hPublicKey, hPrivateKey, hKey;
6040 CK_MECHANISM keyPairMechanism = {
6041     CKM_DH_PKCS_KEY_PAIR_GEN, NULL_PTR, 0
6042 };

```

```

6043 CK_BYTE prime[] = {...};
6044 CK_BYTE base[] = {...};
6045 CK_BYTE publicKeyValue[128];
6046 CK_BYTE otherPublicValue[128];
6047 CK_MECHANISM mechanism = {
6048     CKM_DH_PKCS_DERIVE, otherPublicValue, sizeof(otherPublicValue)
6049 };
6050 CK_ATTRIBUTE pTemplate[] = {
6051     CKA_VALUE, &publicValue, sizeof(publicValue)}
6052 };
6053 CK_OBJECT_CLASS keyClass = CKO_SECRET_KEY;
6054 CK_KEY_TYPE keyType = CKK_DES;
6055 CK_BBOOL true = CK_TRUE;
6056 CK_ATTRIBUTE publicKeyTemplate[] = {
6057     {CKA_PRIME, prime, sizeof(prime)},
6058     {CKA_BASE, base, sizeof(base)}
6059 };
6060 CK_ATTRIBUTE privateKeyTemplate[] = {
6061     {CKA_DERIVE, &true, sizeof(true)}
6062 };
6063 CK_ATTRIBUTE template[] = {
6064     {CKA_CLASS, &keyClass, sizeof(keyClass)},
6065     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
6066     {CKA_ENCRYPT, &true, sizeof(true)},
6067     {CKA_DECRYPT, &true, sizeof(true)}
6068 };
6069 CK_RV rv;
6070
6071 .
6072 .
6073 rv = C_GenerateKeyPair(
6074     hSession, &keyPairMechanism,
6075     publicKeyTemplate, 2,
6076     privateKeyTemplate, 1,
6077     &hPublicKey, &hPrivateKey);
6078 if (rv == CKR_OK) {
6079     rv = C_GetAttributeValue(hSession, hPublicKey, &pTemplate, 1);
6080     if (rv == CKR_OK) {
6081         /* Put other guy's public value in otherPublicValue */
6082         .
6083         .
6084         rv = C_DeriveKey(
6085             hSession, &mechanism,

```

```

6086     hPrivateKey, template, 4, &hKey);
6087     if (rv == CKR_OK) {
6088         .
6089         .
6090     }
6091 }
6092 }

```

6093 5.19 Random number generation functions

6094 Cryptoki provides the following functions for generating random numbers:

6095 5.19.1 C_SeedRandom

```

6096 CK_DECLARE_FUNCTION(CK_RV, C_SeedRandom) (
6097     CK_SESSION_HANDLE hSession,
6098     CK_BYTE_PTR pSeed,
6099     CK_ULONG ulSeedLen
6100 );

```

6101 **C_SeedRandom** mixes additional seed material into the token's random number generator. *hSession* is
6102 the session's handle; *pSeed* points to the seed material; and *ulSeedLen* is the length in bytes of the seed
6103 material.

6104 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
6105 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
6106 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
6107 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE,
6108 CKR_RANDOM_SEED_NOT_SUPPORTED, CKR_RANDOM_NO_RNG, CKR_SESSION_CLOSED,
6109 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

6110 Example: see **C_GenerateRandom**.

6111 5.19.2 C_GenerateRandom

```

6112 CK_DECLARE_FUNCTION(CK_RV, C_GenerateRandom) (
6113     CK_SESSION_HANDLE hSession,
6114     CK_BYTE_PTR pRandomData,
6115     CK_ULONG ulRandomLen
6116 );

```

6117 **C_GenerateRandom** generates random or pseudo-random data. *hSession* is the session's handle;
6118 *pRandomData* points to the location that receives the random data; and *ulRandomLen* is the length in
6119 bytes of the random or pseudo-random data to be generated.

6120 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
6121 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
6122 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
6123 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE, CKR_RANDOM_NO_RNG,
6124 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

6125 Example:

```

6126 CK_SESSION_HANDLE hSession;
6127 CK_BYTE seed[] = {...};
6128 CK_BYTE randomData[] = {...};
6129 CK_RV rv;

```



```

6130
6131 .
6132 .
6133 rv = C_SeedRandom(hSession, seed, sizeof(seed));
6134 if (rv != CKR_OK) {
6135     .
6136     .
6137 }
6138 rv = C_GenerateRandom(hSession, randomData, sizeof(randomData));
6139 if (rv == CKR_OK) {
6140     .
6141     .
6142 }

```

6143 5.20 Parallel function management functions

6144 Cryptoki provides the following functions for managing parallel execution of cryptographic functions.
6145 These functions exist only for backwards compatibility.

6146 5.20.1 C_GetFunctionStatus

```

6147 CK_DECLARE_FUNCTION(CK_RV, C_GetFunctionStatus)(
6148     CK_SESSION_HANDLE hSession
6149 );

```

6150 In previous versions of Cryptoki, **C_GetFunctionStatus** obtained the status of a function running in
6151 parallel with an application. Now, however, **C_GetFunctionStatus** is a legacy function which should
6152 simply return the value CKR_FUNCTION_NOT_PARALLEL.

6153 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_FUNCTION_FAILED,
6154 CKR_FUNCTION_NOT_PARALLEL, CKR_GENERAL_ERROR, CKR_HOST_MEMORY,
6155 CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED.

6156 5.20.2 C_CancelFunction

```

6157 CK_DECLARE_FUNCTION(CK_RV, C_CancelFunction)(
6158     CK_SESSION_HANDLE hSession
6159 );

```

6160 In previous versions of Cryptoki, **C_CancelFunction** cancelled a function running in parallel with an
6161 application. Now, however, **C_CancelFunction** is a legacy function which should simply return the value
6162 CKR_FUNCTION_NOT_PARALLEL.

6163 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_FUNCTION_FAILED,
6164 CKR_FUNCTION_NOT_PARALLEL, CKR_GENERAL_ERROR, CKR_HOST_MEMORY,
6165 CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED.

6166 5.21 Callback functions

6167 Cryptoki sessions can use function pointers of type **CK_NOTIFY** to notify the application of certain
6168 events.

5.21.1 Surrender callbacks

Cryptographic functions (*i.e.*, any functions falling under one of these categories: encryption functions; decryption functions; message digesting functions; signing and MACing functions; functions for verifying signatures and MACs; dual-purpose cryptographic functions; key management functions; random number generation functions) executing in Cryptoki sessions can periodically surrender control to the application who called them if the session they are executing in had a notification callback function associated with it when it was opened. They do this by calling the session's callback with the arguments (hSession, CKN_SURRENDER, pApplication), where hSession is the session's handle and pApplication was supplied to **C_OpenSession** when the session was opened. Surrender callbacks should return either the value CKR_OK (to indicate that Cryptoki should continue executing the function) or the value CKR_CANCEL (to indicate that Cryptoki should abort execution of the function). Of course, before returning one of these values, the callback function can perform some computation, if desired.

A typical use of a surrender callback might be to give an application user feedback during a lengthy key pair generation operation. Each time the application receives a callback, it could display an additional "." to the user. It might also examine the keyboard's activity since the last surrender callback, and abort the key pair generation operation (probably by returning the value CKR_CANCEL) if the user hit <ESCAPE>.

A Cryptoki library is *not required* to make *any* surrender callbacks.

5.21.2 Vendor-defined callbacks

Library vendors can also define additional types of callbacks. Because of this extension capability, application-supplied notification callback routines should examine each callback they receive, and if they are unfamiliar with the type of that callback, they should immediately give control back to the library by returning with the value CKR_OK.

6 PKCS #11 Implementation Conformance

6191

6192

6193

6194

6195

6196

An implementation is a conforming implementation if it meets the conditions specified in one or more server profiles specified in **[PKCS #11-Prof]**.

If a PKCS #11 implementation claims support for a particular profile, then the implementation SHALL conform to all normative statements within the clauses specified for that profile and for any subclauses to each of those clauses .

Appendix A. Acknowledgments

The following individuals have participated in the creation of this specification and are gratefully acknowledged:

Participants:

List needs to be pasted in here

Gil Abel, Athena Smartcard Solutions, Inc.

Warren Armstrong, QuintessenceLabs

Jeff Bartell, Semper Foris Solutions LLC

Peter Bartok, Venafi, Inc.

Anthony Berglas, Cryptsoft

Joseph Brand, Semper Fortis Solutions LLC

Kelley Burgin, National Security Agency

Robert Burns, Thales e-Security

Wan-Teh Chang, Google Inc.

Hai-May Chao, Oracle

Janice Cheng, Vormetric, Inc.

Sangrae Cho, Electronics and Telecommunications Research Institute (ETRI)

Doron Cohen, SafeNet, Inc.

Fadi Cotran, Futurex

Tony Cox, Cryptsoft

Christopher Duane, EMC

Chris Dunn, SafeNet, Inc.

Valerie Fenwick, Oracle

Terry Fletcher, SafeNet, Inc.

Susan Gleeson, Oracle

Sven Gossel, Charismathics

John Green, QuintessenceLabs

Robert Griffin, EMC

Paul Grojean, Individual

Peter Gutmann, Individual

Dennis E. Hamilton, Individual

Thomas Hardjono, M.I.T.

Tim Hudson, Cryptsoft

Gershon Janssen, Individual

Seunghun Jin, Electronics and Telecommunications Research Institute (ETRI)

Wang Jingman, Feitan Technologies

Andrey Jivsov, Symantec Corp.

Mark Joseph, P6R

Stefan Kaesar, Infineon Technologies

Greg Kazmierczak, Wave Systems Corp.

6238 Mark Knight, Thales e-Security
6239 Darren Krahn, Google Inc.
6240 Alex Krasnov, Infineon Technologies AG
6241 Dina Kurktchi-Nimeh, Oracle
6242 Mark Lambiase, SecureAuth Corporation
6243 Lawrence Lee, GoTrust Technology Inc.
6244 John Leiseboer, QuintessenceLabs
6245 Sean Leon, Infineon Technologies
6246 Geoffrey Li, Infineon Technologies
6247 Howie Liu, Infineon Technologies
6248 Hal Lockhart, Oracle
6249 Robert Lockhart, Thales e-Security
6250 Dale Moberg, Axway Software
6251 Darren Moffat, Oracle
6252 Valery Osheter, SafeNet, Inc.
6253 Sean Parkinson, EMC
6254 Rob Philpott, EMC
6255 Mark Powers, Oracle
6256 Ajai Puri, SafeNet, Inc.
6257 Robert Relyea, Red Hat
6258 Saikat Saha, Oracle
6259 Subhash Sankuratipati, NetApp
6260 Anthony Scarpino, Oracle
6261 Johann Schoetz, Infineon Technologies AG
6262 Rayees Shamsuddin, Wave Systems Corp.
6263 Radhika Siravara, Oracle
6264 Brian Smith, Mozilla Corporation
6265 David Smith, Venafi, Inc.
6266 Ryan Smith, Futurex
6267 Jerry Smith, US Department of Defense (DoD)
6268 Oscar So, Oracle
6269 Graham Steel, Cryptosense
6270 Michael Stevens, QuintessenceLabs
6271 Michael StJohns, Individual
6272 Jim Susoy, P6R
6273 Sander Temme, Thales e-Security
6274 Kiran Thota, VMware, Inc.
6275 Walter-John Turnes, Gemini Security Solutions, Inc.
6276 Stef Walter, Red Hat
6277 James Wang, Vormetric
6278 Jeff Webb, Dell
6279 Peng Yu, Feitian Technologies

6280 Magda Zdunkiewicz, Cryptsoft
6281 Chris Zimman, Individual

Appendix B. Manifest constants

The definitions for manifest constants specified in this document can be found in the following normative computer language definition files:

- [include/pkcs11-v3.00/pkcs11.h](#)
- [include/pkcs11-v3.00/pkcs11t.h](#)
- [include/pkcs11-v3.00/pkcs11f.h](#)

Appendix C. Revision History

Revision	Date	Editor	Changes Made
wd01	Apr 30 2013	Chris Zimman	Initial import into OASIS template
wd02	Dec 11 2017	Chris Zimman	Import of approved ballot items
wd05	Nov 14 2018	Tim Hudson	<ul style="list-style-type: none"> - remove C_GetFunctionLists (replaced with C_GetInterfaceList and C_GetInterface) - remove CK_INTERFACES - remove CK_FUNCTION_LISTS - remove MAX_FUNCTION_LISTS - add C_GetInterfaceList using same semantics as C_GetMechanismList - add C_GetInterface using optional CK_VERSION to specific version rather than in the string interface name - add typedefs for the 3.0 function structures - add C_SessionCancel to the CK_FUNCTION_LIST_3_0 structure - it is currently missing from the header file
wd06	Nov 28 2018	Dieter Bong	<ul style="list-style-type: none"> - changed formatting/style of C_nnn function calls in section 5.x from bold text to Heading 3 - some minor format changes, page breaks
wd07	Feb 6 2019	Dieter Bong	<ul style="list-style-type: none"> - Reworded last sentence in section 2, and added reference to header file - Added MESSAGE flags to Table 8, Mechanism Information Flags - Introduced sections for message based signing and message based verification - Split single section with functions for signing and verification into 2 sections, and re-ordered them to signing – message based signing – verification – message based verification - TJH's proposal to rename flag in Table 9, CK_INTERFACE Flags, accepted - Added sample code for message-based encryption
wd08	Mar 26 2019	Daniel Minder	<ul style="list-style-type: none"> - Removed solved comments of Tim Hudson

			<ul style="list-style-type: none"> - Removed C_LoginUser from CK_FUNCTION_LIST since it's a 3.0 function - Switched C_LoginUser and C_SessionCancel in CK_FUNCTION_LIST_3_0 to align with header file - Changed C_GetInterfaceLists to C_GetInterfaceList at some places (5.4.4 - 5.4.6) - Changed comments in C_EncryptMessageFinal sample code to C style - Changed CK_GCM_AEAD_PARAMS to CK_GCM_MESSAGE_PARAMS in C_EncryptMessageFinal sample code - Added CKR_TOKEN_RESOURCE_EXCEEDED to all sign and verify functions except for their Init functions
WD09	Apr 29 2019	Dieter Bong	<ul style="list-style-type: none"> - Updated section Related work - Reference [TLS] updated; references [TLS12] and [RFC 5705] added - Added Dieter Bong as Editor - Updated Citation Format (link still to be updated) - Put year 2019 in Copyright - Section 4.1.3: changed "the three special attributes ..." to "the four special attributes ..."
WD10	May 28, 2019	Tony Cox	<ul style="list-style-type: none"> - Final cleanup of front introductory texts and links prior to CSPRD

6290